Perfect Hashing as an Almost Perfect Subtype Test

ROLAND DUCOURNAU
LIRMM – CNRS and Université Montpellier II, France

Subtype tests are an important issue in the implementation of object-oriented programming languages. Many techniques have been proposed, but none of them perfectly fulfills the five requirements that we have identified: constant-time, linear-space, multiple inheritance, dynamic loading and inlining. In this paper, we propose a subtyping test implementation which involves a combination of usual hashtables and Cohen's display, which is a well known technique for single inheritance hierarchies. This novel approach is based on *perfect hashing*, i.e. an optimized and truly constant-time variant of hashing which applies to *immutable* hashtables. We show that the resulting technique closely meets all five requirements. Furthermore, in the framework of JAVA-like languages—characterized by single inheritance of classes and multiple subtyping of interfaces—perfect hashing also applies to method invocation when the receiver is typed by an interface. The proposed technique is compared to some alternatives, including the proposal by Palacz and Vitek [2003]. Time-efficiency is assessed at the cycle level in the framework of Driesen's pseudo-code and the linear-space criterion is validated by statistical simulation on benchmarks consisting of large-scale class hierarchies.

Categories and Subject Descriptors: D.3.2 [**Programming languages**]: Language classifications—*object-oriented languages*; C++; JAVA; D.3.3 [**Programming languages**]: Language Constructs and Features—*classes and objects*; *inheritance*; D.3.4 [**Programming languages**]: Processors—*run-time environments*; E.2 [**Data**]: Data Storage Representations—*hash-table representations*; *object representations*

General Terms: Experimentation, Languages, Measurement, Performance

Additional Key Words and Phrases: casting, coloring, downcast, dynamic loading, interfaces, method tables, multiple inheritance, multiple subtyping, perfect hashing, single inheritance, subtype test, virtual function tables

1. INTRODUCTION

The need for dynamic subtyping checks is an original feature of object-oriented programming. Given an entity x of a static type C, the programmer or compiler may want to check that the value bound to x is an instance of another type D, generally before applying to x an operation which is not known by C. Three types are involved. D is called the *target* type. The *source* type is the *dynamic* type of x, i.e. the class which has created the value bound to x. C is the *static* type of the expression x. It is usually a supertype of D. However, with multiple inheritance, this is not always the case. Moreover, in a dynamically typed language, x is statically untyped—hence, the type C is the root $\top$ of the type hierarchy. Anyway, a dynamic subtype test determines whether the source type is or is not a

Author's address: R. Ducournau, LIRMM, 161, rue Ada – 34392 Montpellier Cedex 5, France

subtype of the target type. The static type is not involved in the question and there is no need to imply it in the answer. Furthermore, the distinction between class and type does not concern subtyping tests—i.e. in the most common object-oriented framework, C and D are classes and D is a subclass of C (denoted $D \preceq C$).

Subtyping checks may be *explicit* or *implicit*. Programming languages offer various syntactic constructs that enable the programmer to make *explicit* checks. *Downcast* is commonly used in statically typed languages, with the `dynamic_cast` operator in C++ [Stroustrup 1998], parenthesized syntax in JAVA and C# (a C syntax which must not be used in C++!) or *assignment attempts* in EIFFEL [Meyer 1992; 1997]. MODULA-3 [Harbinson 1992] and THETA [Liskov et al. 1995] propose the `typecase` operator. The common point is that the target type is a constant of the construct. Downcast failures may be treated in several ways, either by signaling an exception (JAVA, C++ for references only), returning a null value (EIFFEL, C++ for pointers only), returning `false` (JAVA `instanceof`) or jumping to the next case (`typecase`). When the language is equipped with some *Meta-Object Protocol*—like CLOS [Steele 1990; Kiczales et al. 1991], SMALLTALK [Goldberg and Robson 1983] or JAVA (package `java.lang.reflect`)—the target type may be computed from some other values. For instance, the programmer may want to check that the value bound to x is actually an instance of the class of the value bound to another entity y. Such type checks are handled by boolean methods like JAVA `isInstance`, e.g. `x.isInstance(y.getClass())`. The CLOS functions `typep` and `subtypep` provide similar services. Finally, subtype checks may be *implicit*, when generated by the compiler without being explicitly stipulated by the programmer. This is the case when some language feature is not *type safe*. For instance, array subtyping in JAVA and covariant overriding in EIFFEL are well known unsafe features, which may be valuable [Meyer 1997; Ducournau 2002c] but require dynamic type checking. Implicit subtype checks also occur in exception catch clauses.

The difference between these cases—according to whether the target type is *static* or *dynamic*[1], or the check is *implicit* or *explicit*—is not essential. The only point is that, when the target type is static, the data concerning it may be treated as an immediate value, thus avoiding memory access.

1.1 Perfect hashing

Despite many works from the beginning of object-oriented programming [Schubert et al. 1983; Agrawal et al. 1989; Aït-Kaci et al. 1989; Cohen 1991; Caseau 1993; Habib and Nourine 1994; Habib et al. 1995; Habib et al. 1997; Krall et al. 1997; Vitek et al. 1997; Alpern et al. 2001b; Raynaud and Thierry 2001; Gil and Zibin 2005; Click and Rose 2002; Palacz and Vitek 2003; Alavi et al. 2008], general and efficient implementation of subtype checks remains an open issue.

Without loss of generality, subtype testing amounts to some encoding of the class hierarchy. Many implementations, both naive and sophisticated, amount to searching for the *target* identifier in the data structure of the *source*, which forms a *class ID table*. This is expressed by the following abstract boolean expression:

```
member(targetID,source-classID-table).
```

[1] These terms are unrelated to the C++ operators `static_cast` and `dynamic_cast`—both are *static*.

The search can be naive and linear, or imply data structures like hashtables:

```
hashmember(targetID,source-classID-hashset).
```

An efficient technique proposed by Cohen [1991] replaces this kind of blind search by a single lookup in an array, at a given position, yielding:

```
source-classID-array[targetPosition]=targetID.
```

Unfortunately, like many object-oriented implementation techniques, this is not compatible with both multiple inheritance and dynamic loading, i.e. the original technique is not directly compatible with multiple inheritance and its adaptation to multiple inheritance is not directly compatible with dynamic loading.

In this paper, we present a subtyping test implementation which is fully compatible with both requirements and involves a mixture of hashtables and Cohen's technique. This proposition is based on a 30-year old technique known as *perfect hashing* [Sprugnoli 1977; Mehlhorn and Tsakalidis 1990; Czech et al. 1997], which is an optimized and truly constant-time variant of more traditional hashtables like *linear probing* [Morris 1968; Knuth 1973; Vitter and Flajolet 1990]. Hashtables constitute, together with cache, a common alternative to constant-time implementations of object-oriented languages. However, they are not strictly time-constant. Actually, in the worst case, all the keys may collide on the same hash value, making the search as inefficient as in a naive sequential search. In contrast, perfect hashing optimizes hashtables when they are *immutable*, i.e. when the set of keys is no longer modified after table initialization. This is precisely the subtype testing case, where the hashtable is computed at link or load time and remains immutable at run-time. With perfect hashing, the test takes the following form:

```
source-classID-array[hash(targetID)]=targetID.
```

So the goal of this paper is to present this technique and to study its precise implementation. The exact choice of data structures and code sequences is critical to meet efficiency and functional requirements and will be studied in depth.

1.2 Functional and efficiency requirements

To our knowledge, no implementation strictly meets the following five requirements: (i) truly *constant-time*, (ii) *linear-space*, (iii) compatible with *multiple inheritance*, (iv) compatible with separate compilation and *dynamic loading* and, finally, (v) implemented by a code sequence short enough to be *inlined*.

(i) Constant-time implementation of frequently used mechanisms is a major requirement. Of course, non-constant time implementations may be better on average, but this must be carefully proven—the proof depends on benchmarks or distribution models, which are not always representative. So in this paper we focus on constant-time mechanisms and stress the time-efficiency requirement by using a computational model based on a realistic abstract processor—after Driesen [2001]—that allows a cycle-level analysis. Of course, in a world where thousands of cycles are routinely thrown away upon reflection, security checks, and naive library implementations of data structures, concern about saving a few cycles by instruction-level parallelism may seem obsessive. However, all of these features are implemented using very low level mechanisms such as subtype tests and these basic mechanisms must ensure very high efficiency.

(ii) Object-oriented mechanisms are commonly implemented using tables associ-

ated with classes—the total size of these tables should be linear in the size of the inheritance graph, i.e. linear in the number of pairs (x, y) such that x is a subtype (subclass) of y ($x \preceq y$). In the worst case (i.e. deep rather than broad class hierarchies), this number of pairs is quadratic in the number of classes. Linearity in the number of classes is actually not possible since efficient implementation requires some compilation of inheritance, i.e. some superclass data must be copied in the tables for subclasses. Therefore, usual implementations are, in the worst case, quadratic in the number of classes, but linear in the size of the inheritance relationship. Moreover, the requirement for near-linear use of space implies that the size occupied by a class is linear in the number of its superclasses. In contrast, C++ is, in the worst case, cubic in the number of classes—i.e. all known implementations are cubic and it is likely an inescapable consequence of the language specifications—because the total number of method tables needed for all classes in the hierarchy is exactly the size of the inheritance relationship $\preceq$. The inability to do better than linear-space is likely a consequence of the constant-time requirement. As a counter-example, Muthukrishnan and Muller [1996] propose an implementation of method invocation with $\mathcal{O}(N + M)$ table size, but $\mathcal{O}(\log \log N)$ invocation time, where N is the number of classes and M is the number of method definitions.

(iii) Complex real-world models—such as *ontologies*—require multiple inheritance, at least in the weak form of single inheritance of classes but *multiple subtyping* of interfaces, as in Java and .Net languages like C# [Microsoft 2001]. The fact that few statically typed languages use pure single inheritance, i.e. *single subtyping*, strongly indicates the importance of multiple inheritance. The rare counter-examples, such as Oberon [Wirth 1988; Mössenböck 1993], Modula-3, or Ada 95 [Barnes 1995], result from the evolution of non-object-oriented languages. Furthermore, the absence of Java-like multiple inheritance of interfaces was viewed as a deficiency of the Ada 95 revision, and this feature was incorporated in the next version [Taft et al. 2006]. The requirement for multiple inheritance is less urgent in the dynamic typing framework—for instance, all Java multiple subtyping hierarchies can be directly defined in Smalltalk, by simply dropping all interfaces.

(iv) Dynamic loading was an old feature of Lisp and Smalltalk platforms, before becoming a key requirement of modern runtime systems for the Web, in Java and .Net languages. Actually, a more traditional language like C++ is also quite compatible with dynamic (or incremental) linking. The point is that we only consider techniques that are working under the *open world assumption*—the whole class hierarchy is not presumed to be known when the technique is applied. However, unloading or reloading classes is not part of our requirements, although it should be possible to specify these features in a way that is compatible with the proposed approach. Dynamic loading requires efficient load-time algorithms for encoding the class hierarchy. This can be ensured by incremental algorithms. One might argue—and some authors do, e.g. Palacz and Vitek [2003]—that a fast global (i.e. non-incremental) algorithm may be compatible with dynamic loading, since speed allows complete recomputation at each class loading. Actually, any technique can be used in a dynamic loading framework, at the expense of dynamic data structures, hence extra indirections, and of some recomputation, which may be complete in the worst cases. When the considered technique is inherently non-

Table I. Subtyping techniques compared to functional and efficiency requirements

	constant time	linear space	multiple inheritance	inherently incremental	short code
[Schubert et al. 1983]	+	+	–	–	+
[Agrawal et al. 1989]	–	+	+	–	+
[Cohen 1991]	+	+	–	+	+
[Vitek et al. 1997]	+	+	+	–	+
[Gil and Zibin 2005]	+	+	+	–	+
[Alpern et al. 2001b]	+	–	+	+	+
[Click and Rose 2002]	–	+	+	+	+
[Palacz and Vitek 2003]	+	+	+	–	+
[Alavi et al. 2008]	–	+	+	+	+
C++ (g++)	–	?	+	+	–
SMART EIFFEL	–	+	+	–	–
direct access	+	–	+	+	+
perfect hashing	+	+	+	+	+

incremental, using it in a dynamic framework might make it lose all its desired qualities. Finally, for dynamic loading, the run-time system must not be constrained by some physical limitation, e.g. class number, maximal table size, that would be of the same magnitude as the parameters of actual possible executions. Therefore, all hard-wired limitations should be far from the expected values or the data structures should be resizable to overcome the constraint.

(v) Finally, inlining is a classic optimization that avoids function calls and improves time efficiency at the expense of an increase in code size, which is expected to be slight. It is a key optimization for frequently used basic mechanisms, provided that the inlined code sequence is small and simple enough—i.e. its size is constant and, preferably, the sequence does not involve any function call. In the context of table-based object-oriented implementations, inlining has the advantage of facilitating common subexpression elimination—one access to the method table may serve for several successive operations, for example a subtype test followed by a method invocation. Furthermore, inlining should help the processor to more accurately predict conditional branching at each specific test site. In contrast, the GNU C++ compiler `g++` compiles `dynamic_cast` into a function call.

All five criteria are not independent—memory usage depends on (ii) and (v), time efficient (i) sequences are generally, but not always, inlinable (v). Regardless of other aspects, incrementality of the algorithms (iv) is in favour of time efficiency (i) and code length (v). Indeed, with load-time recomputations, *static* subtyping checks cannot use immediate values but must make some memory access to the data structure of the target type. Furthermore, the object layout or the method table must also make extra indirections. All these extra memory accesses increase the code length, the cycle count and the cache miss risks, since various disconnected memory areas are likely concerned. So, in our analysis, we shall distinguish between inherently incremental techniques and their possible compatibility with dynamic loading. The latter implies, when a technique is not naturally incremental, a precise assessment of its efficiency in a version that is compatible with dynamic loading.

Table I compares known techniques, discussed hereafter, with the five criteria.

Table II. Instruction set of the abstract assembly language [Driesen 2001, p. 193]

`R1`	a register (any argument without `#`)
`#immediate`	an immediate value (prefix `#`)
`load [R1+#imm], R2`	load the word in memory location `R1+#imm` to register `R2`
`store R1,[R2+#imm]`	store the word in register `R1` into memory location `R2+#imm`
`add R1, R2, R3`	add register `R1` to `R2`. Result is put in `R3`
`and R1, R2, R3`	bit-wise `and` on register `R1` and `R2`. Result is put in `R3`
`call R1`	jump to address in `R1` (can also be immediate), save return address
`comp R1, R2`	compare value in register `R1` with `R2` (`R2` can be immediate)
`beq #imm`	if last compare is equal, jump to `#imm` (also `bne`, `blt`, `bgt`, ...)
`bz #imm`	if last operation result is zero, jump to `#imm` (also `bgz`, `blz`)
`jump R1`	jump to address in `R1` (can also be immediate)

1.3 Pseudo-code model and space-simulation for efficiency assessment

Efficiency assessment concerns both execution time and memory usage. We base our evaluation of time efficiency and code size upon an abstract assembly language. This intuitive pseudo-code is borrowed from Driesen and Hölzle [1995], Driesen et al. [1995] and Driesen [2001]. The reader may find more information on its interpretation there, and in Ducournau [2002a]. Table II presents its instruction set. The processor specifications are mostly the same as processor P95 in Driesen [2001]: particularly, 2 `load` instructions cannot be executed in parallel, but need a one cycle delay and the maximum number of integer instructions per cycle is 2. Actually, we should here gain from 3 parallel threads in only one case, of minor importance. The cycle count of each example is given as a function of L and B, which are the respective latencies for memory loads and indirect branching, i.e. mispredicted or unpredictable branching. Typical values are $L = 3$ and $B = 10$. When the cycle count is not straightforward, i.e. when some parallelism is involved, a diagram goes with the code sequence and gives a possible schedule. We exclude cache misses from the cycle count—each one amounts to 100 cycles or even more, but they cannot be considered in our simple framework. However, we examine the number of cache misses that may occur in a given code sequence, i.e. the number of distinct memory areas from which data are loaded. Hence, the final measure is 4-fold—cycle count, code length, cache miss risks and the number of conditional branchings which all entail some misprediction risk, i.e. B-cycle latency. Furthermore, mispredictions can be taken into account in different ways. In common programming style, normal subtype tests are actually undertaken by method invocation. Explicit subtype tests are uncommon—they are presumed to succeed and failure is often managed in an exceptional way. So there is very little chance of mispredicting successes. In this framework, mispredictions must only be considered when there are several logical paths leading to success. On the contrary, abnormal use of subtyping tests, e.g. of JAVA `instanceof` operator, can make failures usual and transform subtype testing into a common control structure. In this case, success and failure might become equally probable. Anyway, we shall not distinguish between the two styles.

Regarding memory usage, we base our evaluation on a simulation of all considered mechanisms on a large set of real-size benchmarks commonly used in the object-oriented language implementation community.

1.4 Plan

The structure of the paper is as follows. Section 2 first presents the well known Cohen [1991] test, which fulfills all requirements but multiple inheritance, and then its extension to multiple inheritance by Vitek et al. [1997], which is no longer incremental. Different notions are introduced on the way. Section 3 describes simple hash-based implementations and shows how *perfect hashing* improves them. The hash-based implementations are applied to C++ object representations in which an object of a class C contains a subobject of each of C's superclasses. The next section examines the case of JAVA and applies perfect hashing to interface target subtype testing and also to method invocation on interface-typed receivers, which is known as the `invokeinterface` operation. Section 5 presents the results of large-scale simulation on a set of benchmarks consisting of large class hierarchies. The choice between two hash functions involves a tradeoff between time and space efficiency. Applications to C++ and JAVA are examined. Section 6 presents other related techniques and discusses how they fulfill—or not—our five requirements. In the conclusion, we examine how this perfect hashing application could be improved, in both its usage and assessment. An Appendix presents the algorithms for computing perfect hashing and discusses benchmark representativeness.

2. MULTIPLE INHERITANCE VS. DYNAMIC LOADING

The simplest subtyping checks work well in single inheritance but either they do not simply extend to multiple inheritance, or their natural extension is no longer incremental, thus less compatible with dynamic loading.

2.1 Single inheritance

2.1.1 *Basic object implementation.* In separate compilation of statically typed languages, late binding is generally implemented with method tables (aka *virtual function tables* in C++ jargon). Method calls are then reduced to function calls through a small fixed number (usually 2) of extra indirections. An object is laid out as an attribute table, with a header pointing at the method table and possibly some added information, e.g. for garbage collection. With single inheritance, the class hierarchy is a tree and the tables implementing a class are straightforward extensions of those of its single direct superclass (Figure 1). The resulting implementation respects two essential *invariants*: (i) a *reference* to an object does not depend on the static type of the reference; (ii) the *position* of attributes and methods in the table does not depend on the dynamic type of the object. Therefore, all accesses to objects are straightforward. This simplicity is due to both static typing and single inheritance—dynamic typing adds the same kind of complication as multiple inheritance, since the same property name may be at different places in unrelated classes.

2.1.2 *Subtype test: Cohen's display.* In the single inheritance framework, many techniques have been proposed. We present one of the most common and simplest ones, which is the basis for further generalizations.

Notations and definitions. $\prec$ denotes the class specialization (aka inheritance) relationship, which is transitive, antireflexive and antisymmetric, and $\preceq$ is its reflexive

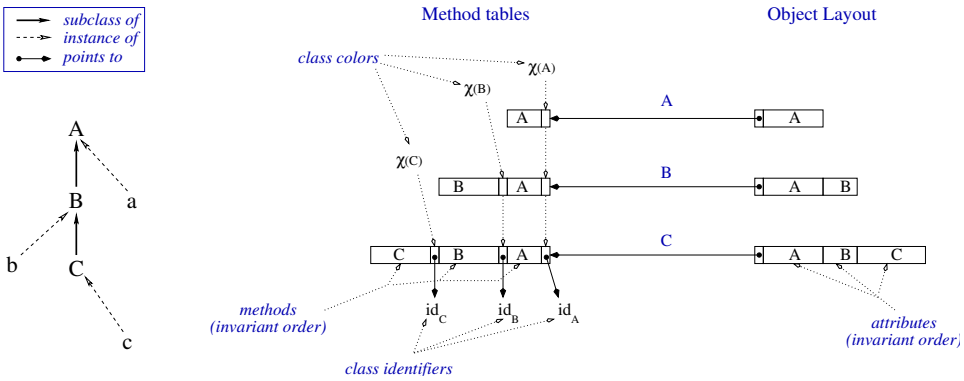


Fig. 1. A small class hierarchy with 3 classes A, B and C with their respective instances, a, b and c. Method tables—including Cohen's display—(left) and object layout (right) in single subtyping. This diagram respects the following conventions. In method tables (left), A (resp. B, C) represents the set of addresses of methods introduced by A (resp. B, C). Method tables are drawn from right to left to reduce edge crossing. In object layouts (right), the same convention applies to attributes but the tables are drawn from left to right. Solid lines are pointers.

closure. Note that we identify here types to classes and subtyping to specialization. Given an object x, $\tau_d(x)$ denotes the class of x (aka its *dynamic type*), i.e. the class which produced x by instantiation. The key characteristic of specialization is that $C \prec D$ implies that all instances of C are instances of D. Therefore, x is an instance of another class C iff $\tau_d(x) \preceq C$. In a static typing framework, the static type of the reference to x is a supertype of the instantiation class. However, contrary to object layout and method tables, the subtype test does not depend on whether typing is static or dynamic.

Principle. This technique was first described by Cohen [1991] as an adaptation of the "display" originally proposed by Dijkstra [1960]. It has been widely reused by different authors (e.g. Queinnec [1998] and Alpern et al. [2001b]). We describe a somewhat improved and partially novel variant (already presented in [Ducournau 2002a]). The technique consists of assigning to each class C a unique identifier id_C together with an offset $\chi(C)$ (called the *color* of C) in the method tables of its subclasses—the corresponding table entry must contain the class identifier. Given an object x of dynamic type $\tau_d(x)$, this object is an instance of a class C iff the object's method table, denoted $tab_{\tau_d(x)}$, contains, at offset $\chi(C)$, the identifier id_C:

$$\tau_d(x) \preceq C \Leftrightarrow tab_{\tau_d(x)}[\chi(C)] = id_C \tag{1}$$

Originally, Cohen's test was implemented in dedicated tables and $\chi(C)$ was the depth of C in the hierarchy tree. However, in a statically typed object-oriented framework, it is more efficient to merge the display within the method tables (Figure 1). Class offsets are then ruled by the same position invariant as methods and attributes and $\chi(C)$ is defined, for instance (assuming that each method pointer and each class identifier occupies one unit of storage), as the number of methods inherited by C plus the depth of C.

The code for testing condition (1) is quite straightforward:

```
  load [object + #tableOffset], table
* load [table + #targetColor], classId
  comp classId, #targetId                                    2L + 2
  bne #fail
  // succeed
```

`object` is the register which holds the object address, `#tableOffset` is the offset of the pointer to method table in the object layout (usually 0), `#targetColor` is the offset of the target class ($\chi(C)$), and `#targetId` the target class identifier (id_C). In terms of processor cycles, the execution of the sequence takes $2L + 2$ cycles, assuming that the processor predicts the right branch. In case of misprediction, B cycles are added. The loads in this example are from two different memory areas and may entail two distinct cache misses. However, since access to the object is mandatory in all the considered implementations, we take only a single extra load into account, which is marked by a '*' in the code sequence.

Two points must be carefully examined. First, when inlining the Cohen display in the method table, a class identifier must not be confused with a method address. As addresses are even numbers (due to word alignment), encoding class identifiers with odd numbers avoids any confusion between the two types. So, if method tables contain only method addresses and class identifiers, both ruled by the position invariant, the value id_C can appear in a table tab_D only at the position $\chi(C)$, when D is a subclass of C: $tab_D[i] = id_C \Rightarrow i = \chi(C)\ \&\ D \preceq C$. Secondly, in theory, $tab_{\tau_d(x)}[\chi(C)]$ is sound only if $\chi(C)$ does not run out of the bounds of $tab_{\tau_d(x)}$. If one assumes that class offsets are positive integers, it would be essential to compare $\chi(C)$ with the length of $tab_{\tau_d(x)}$, together with memory access to the length itself. However, this would add three instructions, one cycle and one misprediction risk and hinder efficiency. Fortunately, there is a simple way to avoid this test, by allocating method tables in a dedicated memory area, which only contains method tables. An extra condition is that this dedicated area—not method tables—must be padded with some even number, to a length corresponding to the maximum *tab* size[2]. This ensures that, in this area, the value id_C cannot occur elsewhere than at offset $\chi(C)$ of the table tab_D of some subclass D of C: $area[i] = id_C \Rightarrow \exists D \preceq C$ such that $i = \chi(C) + addr(tab_D) - addr(area)$, whereby *area* denotes the dedicated area and *addr* returns the address of its argument.

An analogy. An interesting analogy between subtype tests and method calls can be drawn from Cohen's display. Suppose that each class C introduces a method `amIaC?` which returns `yes`. In dynamic typing, calling `amIaC?` on an unknown receiver x is exactly equivalent to testing if x is an instance of C—in the opposite case, an exception will be signaled. In static typing, the analogy is less direct, since a call to `amIaC?` is only legal on a receiver statically typed by C, or a subtype of C—this is type safe but quite tautological. However, subtype testing is inherently type unsafe and one must understand `amIaC?` as a pseudo-method, which is actually

[2]In the dynamic loading framework, this maximum size is not known—an upper bound must be used, which is a parameter of the runtime platform. However, with the dedicated memory area, this physical limitation does not constrain programs since the padded size can be larger than all expected tables without significant space overhead. Moreover, the area size itself is not a limitation—when the current one is full, a new one can be allocated.

not invoked but whose presence is checked. The test fails when this pseudo-method is not found, i.e. when something else is found at its expected position.

This informal analogy is important—it implies that one can derive a subtype testing implementation from almost any method call implementation. We actually know a single counter-example, when the implementation depends on the static type of the receiver, as in subobject-based implementations (Section 2.2.1). The converse does not hold as well, if one wants to closely mimic implementations, but Queinnec [1998] proposes a general method dispatch technique which only relies on subtype testing, irrespective of the precise technique.

Evaluation. Clearly, Cohen's test fulfills all requirements but multiple inheritance: (i) constant-time, (ii) linear-space, (iv) dynamic loading and (v) inlining. Dynamic loading requires some explanations. First, a class identifier must be assigned to each class when it is loaded. Simple numbering is the solution, maybe using only odd or even numbers, or negative ones, according to the chosen encoding. Classes are assumed to be loaded in a natural specialization ordering, i.e. C before D when $D \prec C$. Secondly, all offsets and identifiers (`#targetColor` and `#targetId`) can appear as symbols in the code generated by the compiler. These symbols must be replaced by values at load-time. This is also feasible in a global linking framework, provided that the linker is equipped with the specific but simple functionality for computing colors and identifiers. Colors could also be computed at compile-time but this would unnecessarily increase the recompiling needs.

The space cost is also linear in the sense of criterion (ii) as it is a one-to-one encoding of the specialization partial order, i.e. each specialization pair uses one entry in the tables. A simple improvement consists of assigning two classes to the same word, since short integers are likely sufficient for encoding class identifiers. However, the overall cost of Cohen's test is not negligible. As already stated, this subtype test implementation conceptually adds one method per class, but it needs only a half word per entry. Statistics on method numbers in a set of benchmarks commonly used in the object-oriented language implementation community (see Table IV, page 31) now show that the average number of methods introduced by classes of a benchmark runs, for various benchmarks, between 0.5 and 15. In the latter case, the relative overhead would be insignificant but not in the former, though total sizes are not directly related with the number of introduced methods.

2.1.3 *Variants.* There are several alternatives for implementing Cohen's display.

Variant 1: dedicated or separate class ID tables. A first alternative involves implementing the Cohen display in *dedicated* tables instead of merging it with method addresses—$\chi(C)$ is now the depth of C. These dedicated tables can still be inlined in the method tables, but only at negative offsets since they are size-variable. $\chi(C)$ is then the negation of the depth of C and all the previously described constraints hold, except that the memory area must be padded on the left. Instead of inlining the Cohen display within the method table, it can be implemented in a *separate* table which is pointed by the method table. This requires an extra load. If these separate tables are allocated in their own dedicated area, the bound check can be saved but the extra load entails some cache miss risk. This may be the solution if non-confusing encoding of class identifiers is not possible, e.g. if method tables con-

tain more data than addresses and identifiers, i.e. something that might take any half-word or word value, even though we do not identify what. Note that dynamic typing requires dedicated class ID tables since, in this framework, the method tables cannot benefit from the same simplicity as in static typing—the same method selector might be placed at different positions in two unrelated classes.

Variant 2: uniform-size or fixed-size tables. A more common and frequently proposed way to save on the bound check is to use uniform-size dedicated tables—for all classes C, tab_C has the same size, presumed to not be less than $\chi(C)$. This somewhat amounts to padding each method table, instead of padding only the single—or the very few—memory areas dedicated to method tables. Click and Rose [2002] attribute the technique to Pfister and Templ [1991]. However, the statistics in Section 5 show that, on the same aforementioned benchmarks, the maximum superclass number may be 5-fold greater than its average and, on most benchmarks, the maximum is more than twice the average (Table III). Therefore, uniform size would be at the expense of large space overhead. Moreover, uniform-size tables can be separate or inlined in the method table. Inlining fixes the size. This a significant limitation of the possible executions, since the size becomes a fixed upper bound for the depth of class hierarchies[3]. Hence, oversized tables are the only way to avoid the risk of table overflow at class loading. In contrast, separate tables could be resized when an overflow occurs, but fixed size is no longer justified since the allocation in dedicated area avoids both bound checks and resizing.

Variant 3: non-injective class IDs. When the color of a class is its depth in the hierarchy—i.e. with dedicated class ID tables—more compact identifiers are given by numbering classes on the same level, instead of global numbering, so that the identifiers are unique only among classes with the same depth. Hence, $C \mapsto id_C$ is no longer injective, contrary to $C \mapsto (id_C, \chi(C))$. This may allow us to encode class identifiers with bytes instead of short integers. However, this is at the expense of explicit bound checks or uniform-size tables, since the same identifier can now be at different positions in the same table. Uniform-size tables would however be at odds with the space saving goal. More compact variants exist, with variable length encoding of class identifiers. However, this is to the detriment of dynamic loading, time efficiency and code size, as colors are no longer aligned to bytes.

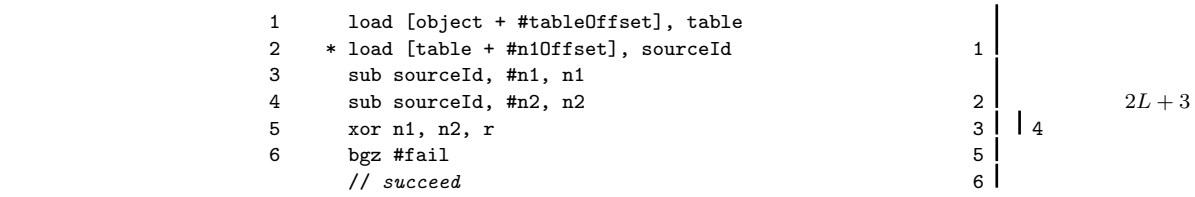
2.1.4 *Schubert's numbering.* This alternative to Cohen's display is a double class numbering, denoted n_1 and n_2: n_1 is a preorder depth-first numbering of the inheritance tree and n_2 is defined by $n_2(C) = \max_{D \preceq C}(n_1(D))$. Then, given an object x:

$$\tau_d(x) \prec C \Leftrightarrow n_1(C) < n_1(\tau_d(x)) \leq n_2(C) \tag{2}$$

This technique, proposed by Schubert et al. [1983], is also called *relative numbering*. It only requires an $\mathcal{O}(N \log N)$ space—in practice, two short integers per class and the first one (n_1) can serve as a class identifier. For the test (2), $n_1(\tau_d)$ is dynamic, whereas $n_1(C)$ and $n_2(C)$ are static when the test is static—they can be compiled

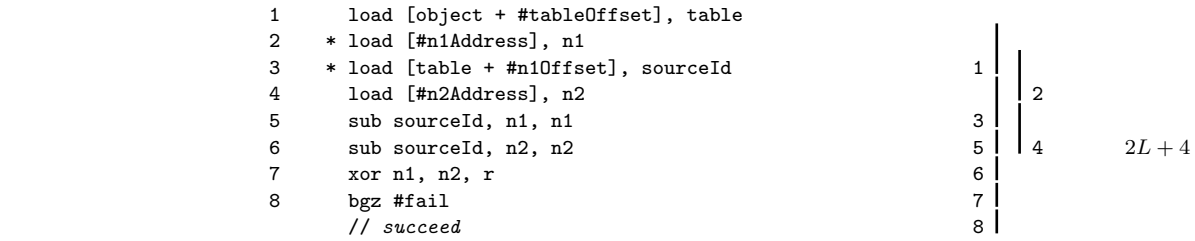
[3]Such an upper bound does not entail the same limitation and overhead, depending on whether it is used for each class, as here, or only for one or a few dedicated areas, as in the previous note. With few dedicated areas, the physical limitation can be oversized without significant overhead.

as constants. As a straightforward implementation would entail two conditional branchings, hence an extra misprediction risk, the condition is transformed according to the following property[4]: $x \in [n_1, n_2]$ iff $\mathbf{xor}(x - n_1, x - (n_2 + 1)) \leq 0$. Then, the code sequence for a static check becomes:



`#n1Offset` is the position of the class identifier in the method table and `sourceId` represents $n_1(\tau_d)$. Some parallelism is possible, which saves on one cycle, and a possible schedule is given by the diagram on the right. `#n1` and `#n2` stand for $n_1(C)$ and $n_2(C) + 1$ and, in a global setting, they can be immediate values.

Variant 4: dynamic loading framework. When used in a dynamic loading framework, this technique requires some recomputation each time a class is loaded—this can be considered since the complexity is linear in the number of classes—and `n1` and `n2` require memory accesses. Hence, the code would be changed into the following:



This adds two `load` instructions but only one cycle as they are run in parallel[5]. However, executing instructions in parallel does not nullify their overhead as they possibly occupy the place of some other instructions. Moreover, the two extra `load`s add one extra cache miss risk, marked by a '*'.

The recomputation which is required at load-time can be eager or lazy [Palacz and Vitek 2003]. Lazy recomputation does not strictly satisfy the constant-time requirement and adds at least one test in the failure case.

2.2 Multiple inheritance

2.2.1 *Standard implementation in static typing.* With multiple inheritance, the two invariants of reference and position, which hold with single inheritance and static typing, cannot hold together, at least when class implementations are sepa-

[4]This property holds for signed positive integers. Warren [2003] proposes a more general condition, which involves an unsigned comparison instead of `xor`.

[5]With processors like Pentium—which allow accesses to subregisters—the two 2-byte `load`s of `n1` and `n2` would be replaced by one 4-byte `load` and subregisters would be used in subsequent instructions. This would save on one instruction, but no cycle in the present case. Of course, this is possible only if short integers are used for both data.

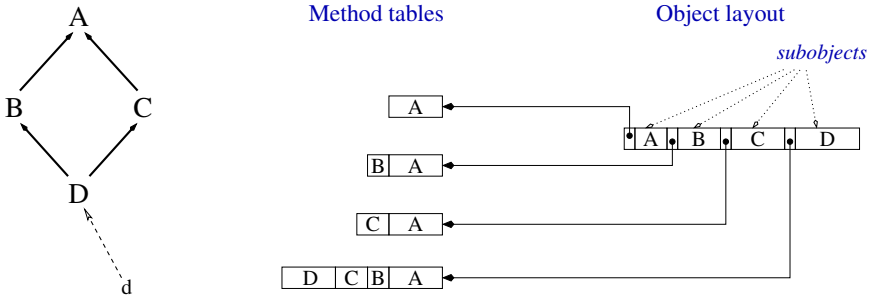


Fig. 2. Object layout and method tables with multiple inheritance: an instance d of the class D is depicted—the diagram follows the same convention as in Figure 1

rately computed, e.g. at load-time. Therefore, the "standard" implementation of multiple inheritance in static typing (SMI in the following)—i.e. that of C++—is based on *subobjects*. The object layout is composed of several subobjects, one for each superclass of the object's class (τ_d). Each subobject contains attributes introduced by the corresponding class, together with a pointer to a method table which contains the methods known—i.e. defined or inherited—by the class (Figure 2). A reference of a given static type points at the subobject corresponding to this type. This is the C++ implementation, when the keyword `virtual` annotates each superclass [Ellis and Stroustrup 1990; Lippman 1996; Ducournau 2002a]. It is time-constant and compatible with dynamic loading, but the number of method tables is equal to the size of $\preceq$, instead of being equal to the number of classes. Accordingly, the total size is no longer linear in the size of the relation $\preceq$, it is linear in the product of relation size and class number and, in the worst case, cubic in the number or classes. Furthermore, all polymorphic object manipulations, which are quite numerous, need pointer adjustments between the static and target types, as they correspond to different subobjects[6].

This implementation has two consequences with respect to our focus of interest. First, the subtyping test is no longer a boolean operation as it must also return, in case of success, the relative position of the target subobject—we shall call it a *shift*, in order to avoid confusion with offsets in the tables. This shift must be added to the address of the static type subobject to get the target address. Therefore, if one considers implementations based on *class ID tables*, the table entries must be twofold: a class identifier and a shift. Secondly, contrary to the analogy on page 9, there is here no known way to derive a subtype test from the technique used for method invocation. It is no longer possible to consider that subtype testing w.r.t. some class C is a kind of `amIaC?` method introduced by C, because this pseudo-method would not have any known position apart from in static subtypes of C. Therefore, one must start from scratch. Of course, one can easily imagine naive techniques for implementing subtype tests, e.g. hashtables. They are, however, presumed to not meet our constant-time requirement. We will examine them in Section 3 but another approach must be examined before ruling it out.

[6]These pointer adjustments are safe—i.e. the target type is always a supertype of the static type—and are implemented more simply than subtyping tests.

Direct access. A technically simple solution allowing constant time is an $N \times N$ direct access matrix, where N is the number of classes: $mat[id_T, id_U]$ contains, if $U \prec T$, the *shift* between subobjects T and U in the object layout of an instance of U, and otherwise a distinguished value for failure. This matrix is global and stands for the whole hierarchy. It requires $2N^2$ bytes, i.e. $2N$ bytes per class, which is a high but yet reasonable cost when there are $N = 100$ classes, but it is not when $N >> 1000$. Class identifiers id_C are computed in the load ordering, hence $id_C \geq id_D$ when $C \preceq D$, and the numbering gives a *linear extension* of $\preceq$. The square matrix *mat* can then be replaced by a triangular one, i.e. by one vector $vect_C$, of size id_C, per class C—then substituting $vect_U[id_T]$ to $mat[id_T, id_U]$. This vector can be inlined in the negative offsets of the method table. The technique is fully incremental. The code for implementing the test is exactly the same as for Cohen's test plus both bound check and pointer adjustment to the τ_d subobject. The average cost is reduced to N bytes per class—this is the worst case of Cohen's technique, when the class hierarchy is a linear chain, but Cohen's display has a far lower practical cost, whereas direct access average always equals its worst-case cost.

In invariant-reference implementations, e.g. in dynamic typing, the vectors could be bit-vectors and the size divided by 16. This would however hardly scale over some hundreds of classes. Clearly, direct access must be ruled out since it would not be space-linear in the sense of criterion (ii). However, with JAVA-like multiple subtyping, the conclusion might be somewhat different (see Section 4).

2.2.2 *Class, attribute and method coloring.* We now present the coloring approach as it is quite versatile and naturally extends the single inheritance implementation to multiple inheritance, while meeting all five requirements except compatibility with dynamic loading. The technique takes its name from *graph coloring*, as its computation amounts to coloring some particular graph[7]. *Method coloring* was first proposed by Dixon et al. [1989] under the name of *selector coloring*. Pugh and Weddell [1990] and Ducournau [1991] applied *coloring* to attributes and Vitek et al. [1997] to classes (under the name of *pack encoding*).

The general idea of coloring is to keep the two *invariants* of single inheritance, i.e. *reference* and *position*. An injective numbering of attributes, methods and classes verifies the position invariant, so this is clearly a matter of optimization for minimizing the size of all tables—or, equivalently, the number of *holes*, i.e. empty entries. However, this optimization cannot be done separately for each class, it requires a global computation for the whole hierarchy. Finally, an important improvement is *bidirectionality*, introduced by Pugh and Weddell [1990], which involves using positive and negative offsets and reduces the hole number. Figure 3 depicts the implementation yielded by unidirectional coloring in the diamond example from Figure 2. The implementation of classes A and B is presumed to be identical to that of Figure 1. Hence, computing the tables for C must reserve some space for B in the tables of D, their common subclass. Thus, some holes appear in the C tables and these holes are filled, in D, by all data specific to B. In contrast, B

[7]This graph is a *conflict graph* with a vertex for each class and an edge between any two vertices that have a common subclass and thus must have their attributes (resp. methods or class IDs) stored at distinct offsets, since attributes (resp. methods or class IDs) of both classes coexist in objects (resp. method tables) of the common subclass.

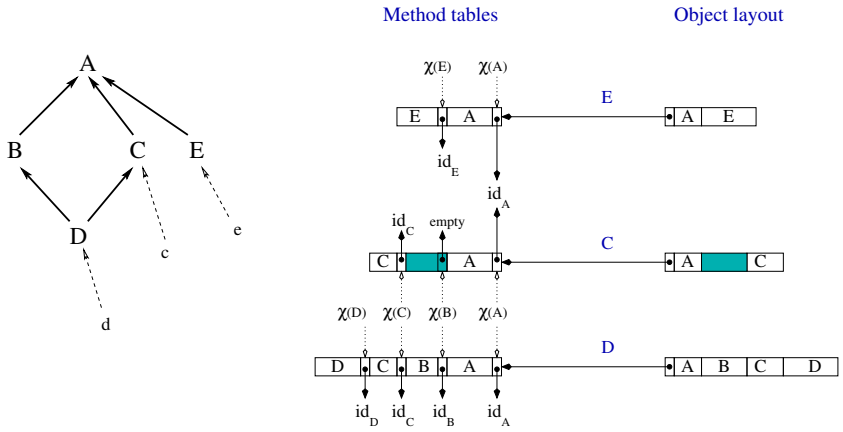


Fig. 3. Unidirectional coloring applied to classes, methods and attributes—the holes (in grey) in the C tables are reserved space for B in the tables of D. A and B classes are presumed to have the same implementation as in Figure 1 and the diagram follows the same convention.

and E have no common subclass and share the same indexes in both method table and object layout. In bidirectional coloring, all holes would have been saved by placing C at negative offsets. The problem of minimizing the total table size is akin to the *minimum graph coloring* problem [Garey and Johnson 1979; Jensen and Toft 1995], which is a well-known problem in the compiler construction area, e.g. for register allocation [Chaitin 1982]. Like minimal graph coloring, the coloring problem considered here has been proven to be NP-hard in the general case [Pugh and Weddell 1993; Takhedmit 2003]. Therefore heuristics are needed and some experiments by Pugh and Weddell [1990], Ducournau [1997; 2002b] and Takhedmit [2003] have shown their efficiency and that the technique is tractable. A detailed presentation of coloring is beyond the scope of this paper and the reader is referred to [Ducournau 2006] which reviews the approach. The point to get is 2-fold: (i) in practice, object layout, method tables and code sequences are exactly those of single inheritance, except for the presence of holes; (ii) this is obtained by rather sophisticated algorithms which require complete knowledge of the class hierarchy. Actually, we have exchanged multiple inheritance for dynamic loading.

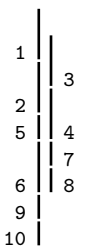
Class coloring is thus the natural extension of Cohen's [1991] technique. Originally described under the name of *pack encoding* by Vitek et al. [1997], it was actually anticipated by the notion of *meet incompatible antichain partition* [Fall 1995]. Regarding the requirements, (v) code sequence and (i) time efficiency are exactly the same as for Cohen's test; (ii) the table sizes are slightly increased (see Section 5), but the overhead is much lower than with subobject-based implementations; but (iii) dynamic loading is not well supported, as recomputations may be required—e.g. when loading D, if B and C have been previously implemented as in Figure 1.

Variant 5: uniform-size tables and non-injective class IDs. In the original proposition by Vitek et al. [1997], the class coloring tables are dedicated and of uniform size, i.e. the minimization criterion is the number of colors, not the overall table

size. Moreover, class identifiers are made non-injective, with their range being possibly restricted to a byte, at the possible expense of increasing the color number. This has, however, the same drawback as for Cohen's test (Variants 2 and 3, page 11), as statistics on large-scale benchmarks show that color tables are more than 2-fold larger when the minimization criterion is the color number instead of the total table size [Ducournau 2006] (see also Tables III and IV). Hence, byte-encoding does not reduce the total size when it is used with uniform-size tables, for this it would require variable-size tables and explicit bound checks.

Variant 6: coloring in a dynamic loading framework. As aforementioned, class coloring can be adapted to dynamic loading at the expense of making all data structures dynamically modifiable. This is basically what is proposed by Palacz and Vitek [2003] for JAVA interfaces. As load-time memory allocation might favor explicit bound checking, we include it in the following version of Cohen's test:

```
 1     load [object + #tableOffset], table
 2   * load [table + #ctableOffset], ctable
 3   * load [#targetColor], color
 4     load [table+#ctableOffset+4], clen
 5     add ctable, color, ctable
 6   * load [ctable], classId
 7     comp color, clen
 8   * bge #fail
 9     comp classId, #targetId
10     bne #fail
       // succeed
```



$3L + 3$

Only `#targetId` remains an immediate value. Access to the color table (`ctable`) and to the target color (`color`) requires an indirection, but the latter can be done in parallel. See also Figure 11, page 27. Above all, two cache miss risks are added, as all data have different locations. The bound check requires 3 (italicized) instructions—they entail one extra misprediction risk.

Precise implementation of color tables and load-time memory management are key issues. Load-time recomputations may force us to enlarge some color tables. Inlined or fixed-size color tables would entail both severe physical limitation and large overhead. Dedicated areas would require specific memory management since they cannot be managed by a common garbage collector. Altogether, the solution relies on separate color tables, which are allocated and garbage collected as any object—hence require an extra word for memory management—with either bound checks or uniform size and possible large space overhead.

3. HASH-BASED APPROACHES

A naive way to implement the subtyping test is via one *hashtable* per class C, associating with the identifier of each superclass of C the shift to the corresponding subobject. In an invariant-reference implementation, the hashtable can be simplified into a *hashset* since all shifts are null. We present hereafter how classic hashtables may be used for subtyping tests, then perfect hashing. We begin in an invariant-reference framework, before discussing subobject-based implementations.

3.1 Linear probing

Among the various hashtable implementations, a simple one is *linear probing* [Morris 1968; Knuth 1973; Vitter and Flajolet 1990]. With linear probing, a hashtable is an array of H entry pairs, alternatively keys in interval $[0..N-1]$ and values, together with a function $h : [0..N-1] \to [0..H-1]$. A key x is sequentially searched in the array, starting from position $h(x)$, until either x is found (success) or a distinguished key $\perp$ (e.g. a negative integer) is found (failure). When the end of the array is found, the search continues at index 0. Experiments and statistical analyses show that the average number of probes is close to 1 when n/H is small enough, where n is the number of occupied entries. A small variant avoids testing the array end by unfolding this circular search, so the array may have more than H entry pairs, up to $H+n$. This improves the code and its efficiency, with minor space overhead. With this variant, H can be less than n, but this would be to the detriment of time efficiency.

In the subtyping test context, such a hashtable is associated with each class C, and n is the number n_C of C superclasses (including C). Therefore, H cannot be uniform for all classes, i.e. the space would be wasted for classes with small n_C/H, and time efficiency would be low when n_C/H is high. Obviously, the constant-time requirement would not be ensured, even on average. Hence, a parameter H_C must be set for each class, for instance $H_C = 2n_C$, and stored in the method table. Consequently, these hashtables are of varying length and can be inlined in method tables at negative offsets only.

A last point must be decided, i.e. the hashing function h. Of course, it depends now on class C, so we need a family of functions h_C, parametrized by H_C, hence $h_C(x) = \mathit{hash}(x, H_C)$. The function *hash* must be simple enough to be inlined—a one-instruction function would be better and a one-cycle instruction the best. Moreover, *hash* is a function from $\texttt{int}_k \times \texttt{int}_k$ to $\texttt{int}_k$—whereby $\texttt{int}_k$ denotes some positive integer implementation, e.g. $\texttt{int}_{16}$ for short integers—such that, $\forall x, y \in \texttt{int}_k, \mathit{hash}(x, y) < y$. Two simple functions are good candidates, i.e. bit-wise `and` and *modulus* (remainder of the integer division, denoted `mod`)[8]. The former seems better as it is a one-cycle instruction, whereas integer division takes several cycles[9].

Finally, with bit-wise `and`, the code for the subtyping test is the following 12-instruction sequence:

```
1         load [object + #tableOffset], table
2       * load [table + #hashingOffset], h
3         and #targetId, h, hv                          2L + 3
4         mul hv, #fieldLen, hv
5         sub table, hv, htable
6  loop:  load [htable + #htOffset], id
```

[8]More precisely, one considers the functions $x \mapsto \texttt{and}(x, H-1)$ and $x \mapsto \texttt{mod}(x, H)$, which both have range $[0..H-1]$.

[9]The latency D of integer division varies markedly according to processors and data, and it may run from some cycles to several tens. For instance, in Intel Pentium IV, integer division uses the micro-coded floating-point unit and takes no less than 20 cycles. In the following, we assume a low hypothetical latency of $D = 6$ cycles. However, an in-depth but partial analysis reveals a 12-25 range for x86 processors, according to the manufacturer and the version. We did not examine other processor families. See also [Warren 2003, Chapter 10].

```
 7         comp #targetId, id
 8         beq #succeed                                    L + 2
 9         comp #empty, id
10       * beq #fail
11         sub htable, #fieldLen, htable
12         jump #loop
 succeed:  // succeed
```

`#hashingOffset` is the offset of the hash parameter ($H_C - 1$) in the method table. The class identifier is hashed and then searched in the hashtable, starting from the hashed value `hv`, until a value `#empty` (i.e. $\perp$) is found. Line 5, the hash value is subtracted, as the hashtable is inlined in the negative offsets of the method table. Line 6, `#htOffset` is the offset of the beginning of the hashtable in the method table—it could be 0. Lines 4 and 11, `#fieldLen` stands for the entry length, which may be 2—assuming that class identifiers are short integers. Line 4, the instruction is actually a 1-cycle shift and it can be saved—it serves for half-word or word alignment but the stored value of bit-wise mask `h` could be already multiplied by `#fieldLen` and, line 3, `#targetId` could be replaced by `#fieldLen*targetId`[10].

There are 12 instructions. This is not very short for inlining and more than 3-fold longer than Cohen's test. The resulting cycle count is $2L + k(L + 5)$, where k is the number of probes, greater than 1 and less than n_C (success) or $n_C + 1$ (failure). Moreover, if the first probe does not succeed, all subsequent conditional branchings add extra misprediction risks, i.e. $2(k - 1)$ in case of success, one more in case of failure[11]. On average, k should be close to 1, so it would be almost 2 times slower than Cohen's test (with $L = 2$ or 3). There is only one cache-miss risk if one assumes that the table is not too large. With modulus, the code would be exactly the same, except that integer division would be substituted for bit-wise `and`, but the corresponding latency adds $D - 1$ cycles.

3.2 Perfect hashing

These classic hashtables are efficient on average, but they are not time-constant and their worst case is quite inefficient. Moreover, they do not take a key particularity of the present application into account, namely that the considered hashtables are *immutable*—once they have been computed, at link or load time, there is no longer any need for insertion or deletion. Therefore, for each class C, knowing the identifiers of all its superclasses, it is possible to optimize the hashtable in order to minimize the table size and the average number of probes, either in positive or negative cases. In the ideal case from a time standpoint, H_C may be defined in such a way that all tests only need one probe, i.e. h_C is injective on the set of identifiers of all superclasses of C. This is known as a *perfect hashing* function [Sprugnoli 1977; Mehlhorn and Tsakalidis 1990; Czech et al. 1997]. Note that we are not searching a *minimal perfect hashing function*, i.e. a function such that $H_C = n_C$, because such

[10] `#fieldLen*targetId` denotes the immediate value which results from the arithmetic expression, not the expression itself. The alternative of `#fieldLen`-multiple class identifiers might prevent us to code class identifiers with short integers, if there are too many classes (see Section 5). However, as an immediate value has only a limited number of bits—from 8 to 16, according to architectures [Driesen 2001]—line 3 might require two instructions.

[11] Actually, the loop should entail at least one misprediction, unless it is exited at the first probe. With conditional branching prediction, short loops are somewhat less efficient than long loops.

```
1      load [object + #tableOffset], table
2    * load [table + #hashingOffset], h
3      and #targetId, h, hv
4      mul hv, #fieldLen, hv
5      sub table, hv, htable                                  3L + 5
6      load [htable +#htOffset], id
7      comp #targetId, id
8      bne #fail
```

Fig. 4. Code sequence for perfect hashing in reference-invariant implementations

a function would be more complicated to both compute and inline.

On the whole, our usage of perfect hashing is quite generic, i.e. it works for all possible *hash* functions, at least when they represent a 'good' hashing function. For each class C, we define H_C as the least positive integer such that h_C—defined as $h_C(x) = hash(x, H_C)$—is injective on the set $I_C = \{id_D \mid C \preceq D\}$ of identifiers of the superclasses of C. When loading class C, its superclasses are first recursively loaded, if needed. Its identifier id_C is then assigned by a simple injective numbering, so that $C \prec D$ implies $id_C > id_D$ and $id_C = \max(I_C)$. Finally, H_C is computed. In the case of `mod` and `and`, this computation is straightforward (see Appendix A).

The linear probing code (page 17) still works but it can be simplified into the 8-instruction sequence presented in Figure 4. The cycle count, $3L + 5$, is almost the same as with linear probing when there is only one probe. The difference is that with perfect hashing there is always one probe. Moreover, the code sequence is quite a bit shorter.

Regarding space, the table size is exactly H_C. Clearly, $H_C \geq n_C$ for all hash functions. Moreover, $H_C \leq id_C + 1$ for `mod`, while $H_C \leq 2^{\lceil \log_2(id_C+1) \rceil} \leq 2id_C$ for `and`, and these upper bounds are worst cases. For instance, with `and`, if $id_C = 16$ and $I_C = \{0, 1, 2, 4, 8, 16\}$, then $H_C = 32$—this follows from the fact that all 1-bit in I_C numbers are required (see the discussion of discriminant bits in Appendix A.2). Of course, these upper bounds are not good—actually, they are not better than direct access. To prove that the average size meets our linear-space requirement, we did not conduct an in-depth mathematical analysis, but instead used simulations on large-scale benchmarks (see Section 5).

Irrespective of this analysis, we shall first examine how to apply this technique if it eventually appears feasible. The previous code sequences could apply to all object-oriented languages with multiple inheritance, dynamic loading and invariant-reference implementation, e.g. in dynamic typing frameworks. Typical examples are CLOS [Steele 1990] or DYLAN [Shalit 1997].

3.3 Application to subobject-based implementations

In the subobject-based implementation framework, hashsets are no longer sufficient and true hashtables are required—a shift must be associated with the searched class identifier in order to adjust the pointer. Therefore two instructions are required for loading the shift and adding it to the pointer.

```
load [htable + #htOffset+fieldLen], delta
add object, delta, object
```

In the linear probing sequence (page 17) and in the perfect hashing case (Figure 4), they must be respectively inserted after instruction 6 and at the end. In terms

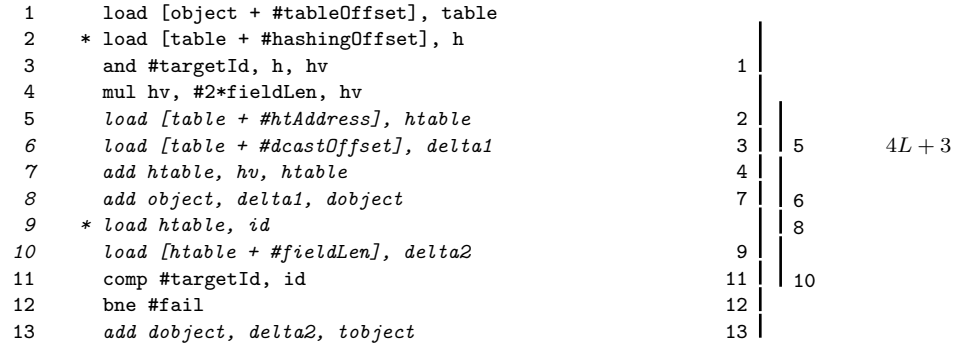


Fig. 5. Perfect hashing in subobject-based implementation—final code sequence

of cycles, in both cases, the extra `load` is done in parallel, irrespective of possible failure, saving on L cycles (see also note 5, page 12), and only one cycle is added.

However, subobjects raise another problem. The previous code sequence presumes that each subobject has its own hashtable in its method table. This would be space expensive, i.e. cubic in the number of classes, and a single hashtable per dynamic type must be preferred. A first solution amounts to inlining the hashtable in the method table of the dynamic type and beginning with a pointer adjustment to this dynamic type, which adds the following 3-instructions and $2L + 1$ cycles at the beginning of the sequence:

```
load [object + #tableOffset], table
load [table + #dcastOffset], delta                    2L + 1
add object, delta, object
```

`#dcastOffset` is the offset of the shift to the dynamic type subobject. However, this does not utilize possible parallelism and the total cycle count is $5L + 7$.

A better solution, that we present only in the case of perfect hashing, is as follows. All the method tables of the considered dynamic type point at a single hashtable, but each one contains the H_C parameter—this will improve parallelism. Figure 5 presents the final code sequence—the instructions which differ from Figure 4 are italicized—and Figure 6 depicts the corresponding layout with all symbols appearing in the code. `#htAddress` is the offset of the hashtable address—it replaces `#htOffset` but requires a separate instruction. `dobject` is the pointer to the subobject corresponding to the dynamic type and `tobject` that of the target type. The resulting code has the same length (12 instructions), with only $4L+3$ cycles in case of success. Note that the cost of dynamic hashing (lines 2-3) is only one cycle, since it is done in parallel with pointer adjustment. However, this is no longer true with modulus, as the integer division latency increases the duration of dynamic hashing by several cycles that cannot be done in parallel.

Regarding space, the hashtables required by subobject-based implementations have the same number of entries as the hashsets which serve for invariant reference. However, each hashtable entry is 2-fold, i.e. class identifier and shift, hence doubling the total size.

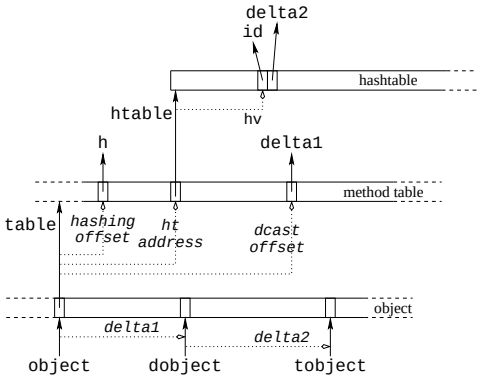


Fig. 6. Perfect hashing in subobject-based implementation (SMI)—object layout, method and hash tables. Pointers and pointed values are in roman type with solid lines, and offsets are italicized with dotted lines.

3.4 Perfect hashing variants

Perfect hashing with `mod` or `and` functions is surely close to optimum from the code length standpoint and, in the case of `and` only, from the cycle count standpoint. However, one should expect that this is no longer optimal from the space standpoint. Therefore, it is also essential to examine alternatives which would reduce the table size, presumably to the detriment of time and code length.

3.4.1 *Quasi-perfect hashing.* A first alternative is *quasi-perfect hashing* [Czech 1998]. Whereas perfect hashing involves a function h which is injective on the considered set of keys I, quasi-perfect hashing involves a pair of hashing functions h_1 and h_2, such that if a key $x \in I$ is not found at place $h_1(x)$ in the first table, it will be found unambiguously at place $h_2(x)$ in the second table. A variant consists of a single function $h : \texttt{int}_k \to [0..H-1]$ such that, $\forall x \in [0..H-1]$, $h^{-1}(x) \cap I$ has at most 2 elements. There are at least two possibilities, $h_1 = h$ and $h_2 = h + H$ or $h_1 = 2h$ and $h_2 = 2h + 1$. In all cases, the total size is $2H$ and we shall prefer the latter because adding 1 is simpler than adding H.

The code for invariant-reference implementations (Figure 4) is transformed by doubling `#fieldLen` (line 4) and replacing the conditional branching to failure (last line) by the following:

```
 8         beq #succeed
 9         load [htable + #htOffset-1], id
10         comp #targetId, id
11       * bne #fail
  succeed: // success
```

This somewhat amounts to unfolding the linear probing loop. The extra 3-instruction sequence can be run in parallel and theoretically adds only one cycle. Subobject-based implementations would require adding two loads after lines 6 and 9, but these loads could not run in parallel, at least in our 2-thread computational model—here is the only aforementioned exception. The extra conditional branching also adds a misprediction risk, i.e. B-cycle latency when the prediction fails. In an invariant-reference implementation, it can be replaced by two subtractions and one multiplication, at the expense of a few more cycles.

Another variant involves keeping the original `#fieldLen` value (line 4). The resulting code works for linear-probing hashtables such that all $x \in I_C$ is found at position $h_C(x)$ or $h_C(x) + 1$, where h_C is now defined by $h_C(x) = \texttt{and}(x, H_C - 2)$

and H_C is the least integer which satisfies the property. Intuitively, the resulting table sizes of both variants should not significantly differ, but in practice random tests are in favor of the latter and the former is markedly better on our benchmarks.

3.4.2 *Partial perfect hashing.* One can encode only the reflexive reduction $\prec$, instead of the full partial order $\preceq$, at the expense of testing the source and target IDs for equality. The resulting code is very close to that of quasi-perfect hashing. Alternatively, one can remove from the hashed set the hierarchy root instead of the source. This is more efficient since the root is a meaningless target in a *static* subtype test—so the inlined code for static tests would remain unchanged. On the contrary, an extra equality test is required when the test is dynamic.

3.4.3 *Two-parameter hashing function.* In the literature on perfect hashing [Czech et al. 1997], perfect hashing functions usually have more than one parameter. Adding a parameter also adds an extra `load`—which would run in parallel, hence without extra cycle—plus an extra operation. If both operations are 1-cycle, the time overhead will be low. However, a second integer division—this gives a classic perfect hashing function [Sprugnoli 1977]—would markedly degrade the efficiency.

The code for invariant-reference implementations (Figure 4) is transformed by replacing instructions 2-3, which implement perfect hashing, by the following sequence, for an hypothetical 2-parameter hashing function:

```
2a      * load [table + #hashingOffset1], h1
2b        load [table + #hashingOffset+fieldLen], h2
3a        op1 #targetId, h1, hv
3b        op2 hv, h2, hv
```

It adds two instructions but only one cycle, as the second `load` can be done in parallel. Compared to quasi-perfect hashing, it has the same cycle count but two instructions less and only a single test, i.e. one misprediction risk.

Of course, one issue remains—namely determining the two 1-cycle operations `op1` and op2 such that $h_C(x) = \texttt{op2}(\texttt{op1}(x, H_C^1), H_C^2)$ is an efficient perfect hashing function, i.e. the resulting table size H_C is small enough.

3.4.4 *Perfect numbering.* So far, we considered that class identifiers were a fixed input and the point was to define compact perfect hashing functions. It is however possible to somewhat invert the problem by computing the best class identifiers that yield the most compact hashtables. So, when loading class C, this new problem is now defined as follows. Given the set $I'_C = \{id_D \mid C \prec D\}$ of identifiers of the *proper* superclasses of C, compute id_C in a way that minimizes H_C, which is itself defined as previously. Let us call this technique *perfect numbering*. Of course, id_C must differ from the identifiers of all the already loaded classes. Furthermore, id_C must be kept small enough to remain in the chosen $\texttt{int}_k$ integer implementation. For instance, id_C might be defined as the least not yet allocated integer that minimizes H_C. However, this does not necessarily minimize $\sum_C H_C$.

4. APPLICATION TO JAVA

As aforementioned, JAVA and .NET languages have a key particularity—i.e. classes are in single inheritance and interfaces in multiple subtyping. Dynamic types are always classes. As both classes and interfaces can serve as static types, subtyping

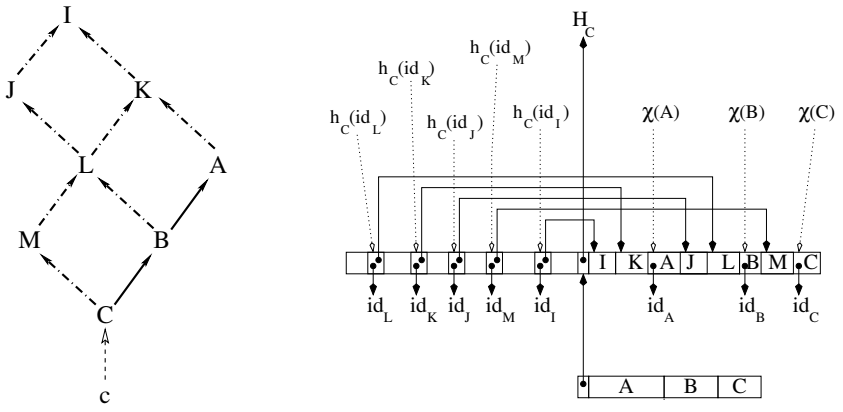


Fig. 7. A JAVA-like, single inheritance and multiple subtyping hierarchy, with 3 classes A, B, C, 5 interfaces, I, J, K, L, M and an instance c. Object layout is the same as in single inheritance. The method table is twofold. Positive offsets involve the method table itself, organized as with single inheritance. Negative offsets consist of the hashtable, which points, for each implemented interface, to the group of methods introduced by the interface. The diagram follows the same convention as in Figure 1.

test and method invocation differ according to whether the considered static type—i.e. the target type or the receiver's static type—is a class or an interface. Perfect hashing applies to both mechanisms when the considered type is an interface.

4.1 Subtype test

When the target type is a class, Cohen's technique applies and there is no need for perfect hashing, which is more expensive. When the target type is an interface, perfect hashing applies, in the invariant-reference form (Section 3.2). In terms of cycles, this is only less than two times slower than Cohen's test, without extra risk. Regarding space, interfaces are only a subset of the whole type hierarchy[12] and hashtables contain only interface identifiers. So one may expect reasonable sizes.

Finally, with *dynamic* tests, when the target type is not statically known, a function must be called, which applies the technique corresponding to the kind of target type. This could also be inlined, but the sequence would be markedly longer.

4.2 Perfect hashing for method invocation

When the receiver is typed by a class, method invocation is the same as in single inheritance. When the receiver is typed by an interface—i.e. the JVM `invokeinterface` operation [Meyer and Downing 1997]—the position invariant does not hold and perfect hashing can be used instead. There are at least two solutions for applying it, i.e. hashing interface or method identifiers.

4.2.1 *Interface method tables.* A first solution is a hashtable associating, with each interface implemented by a class, its own method table that contains only

[12]In the benchmarks we have studied, there are many fewer interfaces than classes but there are also programming styles for which this not the case. Moreover, compilers may generate many interfaces. For instance, in SCALA, each class implies an interface definition [Odersky et al. 2008].

```
1                       load [object + #tableOffset], table
2                     * load [table + #hashingOffset], h
3                       and #interfaceId, h, hv
4                       mul hv, #2*fieldLen, hv
5                       sub table, hv, htable
6 | load [htable +#htOffset-fieldLen], id        | load [htable +#htOffset], itable
7 | comp #interfaceId, id                        | load [itable +#methOffset], method
8 | bne #fail                                    | call method
                      3L + 5                                  4L + B + 3
```

Fig. 8. Final code sequences for JAVA interface implementation, with subtype testing (left) and method invocation (right).

methods introduced by the interface. Rather than an extra method table, this is merely the address of the group of methods introduced by the interface[13] within the class method table—so no extra space is needed (Figure 7). In the particular case of method invocation and perfect hashing, static typing makes the test useless since the search for the interface is always successful. Therefore, the hashtable contains only pointers at interface method tables and the code becomes:

```
1       load [object + #tableOffset], table
2     * load [table + #hashingOffset], h
3       and #interfaceId, h, hv
4       mul hv, #fieldLen, hv
5       sub htable, hv, htable                                  4L + B + 3
6       load [htable + #htOffset], itable
7       load [itable + #methOffset], method
8       call method
```

Extra instructions w.r.t. single inheritance are italicized. `Itable` is the address of the group of methods introduced by the considered interface and `#methOffset` is the offset of the considered method in this group. Here, `#fieldLen` is at least 4, since `itable` is an address. The cycle count is $4L + B + 3$, while single inheritance method invocation takes only $2L + B$. The two extra `load`s do not entail any cache miss risk, as the loaded address is in the already cached method table.

Finally, the same hashtable must serve for both method invocation and subtype testing, as only one table can be inlined at negative offsets in the method tables, since they are of variable length and `#htOffset` is a constant. Merging them is easy as they have the same keys and the first one only requires class identifiers, whereas the second only requires addresses. In that way, the complete code sequences for the two mechanisms are given in Figure 8, where the first five lines are factorized. Figure 9 depicts the corresponding pointers and offsets in the method table. Combining both mechanisms, e.g. invoking a method after having checked that the call is type safe, requires only 11 instructions and $4L + B + 4$ cycles.

Variant 7: offsets vs. addresses. As both original tables respectively have word and half-word entries—assuming that interfaces are few enough to be identified by short integers—merging them would paradoxically increase the total size if word-

[13] A JAVA specification flaw makes it possible for a method signature to be introduced by several interfaces. So, the method must be implemented in each interface group, but the corresponding table entries contain the same method address. When compiling such a method call, the compiler can choose any of the introducing interfaces together with the corresponding offset.

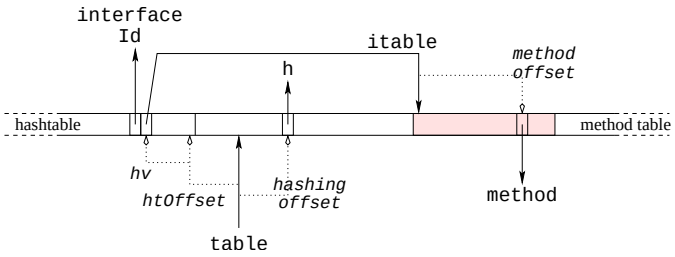


Fig. 9. Perfect hashing of JAVA interfaces. The grey rectangle denotes the group of methods introduced by the considered interface.

alignment is required for efficiency, i.e. `fieldLen`=4. However, replacing the address of interface method tables by their offset in the class method table is a simple way to save space. In that way, half-words are enough, at the expense of an extra `add`, and line 6 is replaced by the two following instructions:

```
6a    load [htable + #htOffset], ioffset
6b    add table, ioffset, itable
```

where `ioffset` is the offset of the considered group of methods—here relative to `table` but it could also be relative to `htable`—and `fieldLen` is 2.

Allocation in dedicated areas. Finally, one must check that the allocation of method tables in dedicated areas is still possible for saving bound checking in Cohen's test. Each hashtable entry is a pair of an interface ID (any positive $\texttt{int}_{16}$) and an offset (any $\texttt{int}_{16}$ that is a positive multiple of 4). Each method table entry is either an address (any $\texttt{int}_{32}$ that is a positive multiple of 4) or a pair of unambiguous class IDs (any $\texttt{int}_{16}$ that is a negative non-multiple of 4). The H_C parameter itself can be any positive $\texttt{int}_{16}$. So a class ID cannot be confused with any half-word in the table.

Perfect-hashing variants. Some variants could also be adapted to method invocation. *Quasi-perfect hashing* would however introduce a test in method invocation, since the expected method address may be in any of the two tables. Therefore, besides a longer sequence with the same apparent cycle count, quasi-perfect hashing would add some misprediction risk. In contrast, *partial perfect hashing* is meaningless here, since neither the source nor the root are member of the hashed set.

In the multiple subtyping framework, *perfect numbering* (Section 3.4.4) may be less suitable, since the identifier of the loaded class is not involved in the optimization. However, instead of numbering interfaces one-by-one as they are loaded, perfect numbering could be applied to a whole set of interfaces that are simultaneously loaded when a class is loaded. This is however a matter of further research.

4.2.2 *Hashing method identifiers.* An alternative is to hash method identifiers instead of interface identifiers. This implies that method identifiers are generated at load time, in the same way as class identifiers, by a simple numbering scheme.

```
1      load [object + #tableOffset], table
2    * load [table + #hashingOffset], h
3      and #methodId, h, hv
4      mul hv, #fieldLen, hv                    3L + B + 4
```

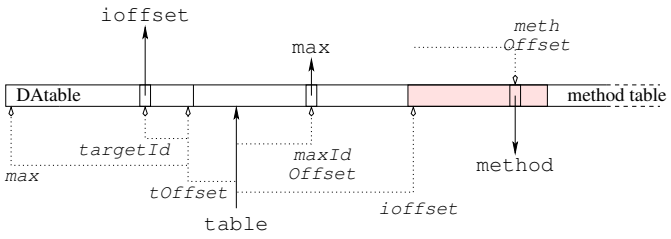


Fig. 10. Direct access to JAVA interfaces

```
5       sub table, hv, table
6       load [table + #htOffset], method
7       call method
```

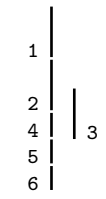
Here, `fieldLen`=4. The advantage is to save one `load` (line 6 in the previous code) to the detriment of space, since all methods introduced by an interface must be hashed. Since there is no way to inline two hashtables in negative offsets, both hashtables—interface hashset for subtyping test and method hashtable for method calls—must be merged. Hence, method and interface identifiers must differ, e.g. respectively even and odd, or with a single global numbering. In this case, there is only one entry per interface (resp. method), its identifier (resp. address), since method invocation is type safe.

4.3 Alternatives to perfect hashing

4.3.1 *Direct access to interfaces.* In Section 2.2.1, we ruled out direct access tables as they are too large—the average space associated with a class C would be linear in the total number of classes (N) instead of being near-linear in the number n_C of superclasses of C. However, with single inheritance and multiple subtyping, there is only a need for direct access to interfaces. Let N_c be the class number, N_i the number of interfaces and $N = N_c + N_i$. The total space would be $N_c \times N_i$, instead of N^2—so, never greater than $N^2/4$, and likely less if interfaces are significantly fewer than classes (see also note 12, page 23). Therefore, a precise assessment requires an examination of direct access.

As for hashtables, the direct access array is inlined at negative offsets. Each entry is either empty (`#empty`) or contains the offset of the interface method table (`ioffset`). The array size, i.e. the maximum identifier (`max`) of implemented interfaces must also be stored in the method table. `#targetId` must be lesser than `max` and `ioffset` must not be `#empty`, but a straightforward implementation would entail an extra misprediction risk. So, assume that `max` is a strictly positive integer, `#empty` is $2^{15} - 1$, i.e. the highest positive short integer, and `ioffset` is a strictly negative offset or `#empty`. The subtyping test code is then:

```
1       load [object + #tableOffset], table
2     * load [table + #maxidOffset], max
3       load [table - #targetId*fieldLen+tOffset], ioffset
4       sub #targetId, max, max
5       and max, ioffset, max
6       bgz #fail
        // succeed
```



$2L + 3$

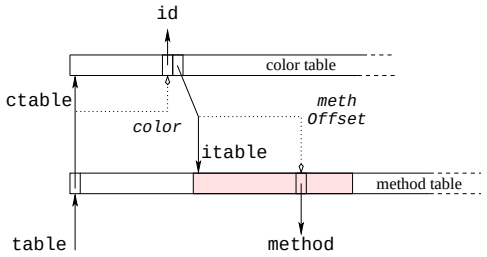


Fig. 11. Incremental coloring of JAVA interfaces

`#tOffset` is the analogue of `#htOffset`. Regarding method invocation, no test is required and the following code is obtained:

```
1      load [object + #tableOffset], table
2    * load [table - #targetId*fieldLen+tOffset], ioffset          3L + B + 1
3      sub table, ioffset, itable
4      load [itable + #methOffset], method
5      call method
```

Figure 10 is a diagram of the corresponding data structure. In both mechanisms, the code lengths and cycle counts are very efficient. An improvement would be to truncate the array to the minimum interface identifier implemented by the class. This implies storing the minimum and making one addition, together with one test in the subtyping case. On the whole, this would give 11 instructions and $3L + 3$ cycles in the subtyping code, and 6 instructions and $4L + B + 1$ cycles for method invocation.

4.3.2 *Incremental coloring (IC).* As aforementioned (Variant 6, page 16), coloring is inherently non-incremental but could be used in a dynamic framework at the expense of possible complete recomputation and more dynamic structures. Applied to method invocation on interface-typed receivers, this amounts to coloring interfaces and associating the interface color plus the interface table address with each entry. Subtype test is as page 16, and method invocation as follows:

```
1      load [object + #tableOffset], table
2    * load [table + #ctableOffset], ctable
3    * load [#interfaceColor], color
4      add ctable, color, ctable                                   4L + B + 1
5    * load [ctable+#fieldLen], itable
6      load [itable + #methOffset], method
7      call method
```

The usual method table (`table`) points to a color table (`ctable`) which is recomputed each time a class loading requires it. In this JAVA framework, coloring has the same 2-fold entries as perfect hashing, but 2 extra cache miss risks are added.

4.3.3 *Shared separate interface-offset tables.* Compared to plain multiple inheritance, the specificity of multiple subtyping is that the considered class is not included in the table that must be built. In other words, this is a partial encoding of $\prec$, not a full encoding of $\preceq$. Therefore, instead of inlining the interface table in the negative offsets of the method table—priority to time—this table could be shared between several classes—priority to space. More precisely, when a class does not

directly implement any interface, it inherits the interface table of its single direct superclass. This entails no overhead at all for incremental coloring whose tables are already separate. In the case of perfect hashing and direct access, this is at the expense of an extra `load`—hence, an extra cache miss risk—but this `load` is done in parallel for perfect hashing (after line 2, Figure 8) and adds only $L-1$ cycles for direct access—on condition that H_C parameters or the size of DA tables remain in the method tables, like in subobject-based implementation. Of course, sharing is only possible in the offset variant, when the considered table contains offsets instead of addresses—indeed, these offsets satisfy the position invariant. Moreover, it is only efficient when there are relatively few interfaces. In contrast, in SCALA, each class implements a new interface, hence there would be no sharing at all.

5. SPACE EFFICIENCY—EXPERIMENTS AND STATISTICS

We assessed the space efficiency of perfect hashing on several large benchmarks commonly used in the object-oriented community implementation, e.g. by Vitek et al. [1997] and Gil and Zibin [2005]. They are abstract schemata of large class libraries, from various languages: JAVA (from the IBM-SF benchmark to the JDK.1.0.2 benchmark, by decreasing size of the specialization relationship, in Table III), CECIL (Cecil, Vortex3), DYLAN, CLOS (Harlequin), SELF, EIFFEL (SmartEiffel) or EIFFEL-like language (Lov and Geode) and PRM [Privat and Ducournau 2005; Privat 2006]. Here, all benchmarks are multiple inheritance hierarchies and are discussed in Appendix B. In the past few years, we have developed an experimental platform—written in COMMON LISP and CLOS—for simulating and comparing various implementation techniques by computing spatial parameters [Ducournau 2002a; 2006]. The following statistics result from a direct extension of this platform. Algorithms for computing H_C values are presented in Appendix A. They are straightforward and the computation time is quite insignificant on all benchmarks. Class identifiers are generated by a simple numbering of classes, in a possible class loading ordering—i.e. an arbitrary *linear extension* (aka *topological sorting*) of $\prec$.

5.1 Application to plain multiple inheritance

5.1.1 *Class hierarchies and subtype testing.* Table III first presents two objective data, i.e. the number n_C of superclasses of each class C, including itself, (first columns) and the number N of classes (last column). Note that N is always less than 2^{14}—this confirms that short integers can be envisaged as class identifiers but this remains an important matter of design[14]. The cons are that the ever-growing size of programs will yield, in the near future, larger class hierarchies. However, the pros are that these benchmarks are class libraries, whose classes are not intended to be all loaded together in a single run, and there is no indication on the maximal number of classes of a given run. Palacz and Vitek [2003] present statistics in favor of this more optimistic thesis. They note that their benchmarks are running JAVA programs based on almost ten-thousand class libraries but that each run hardly

[14]More precisely, as hashing does not require any specific coding of class identifiers, short integers are enough for 2^{16} classes. With Cohen's test and class coloring, the coding is restricted to non-4-multiples, and short integers are limited to 3.2^{14} classes. Finally, if the point is to place two classes on a single word, identifiers must be negative and the number of classes is limited to 3.2^{13}.

Table III. Statistics on class hierarchies and subtype test: superclass number (n_C) and table size for bidirectional class coloring (COL_2), perfect hashing (PH) and quasi-perfect hashing (qPH)—average and maximum per class—compared to class number (N). The 'total' column is the size of the specialization relationship ($\preceq$), i.e. the sum of the parameter n_C for all classes (also the X-coordinate of diagrams in Figure 12). All other numbers are class or entry counts that are comparable and presented in increasing order, from left to right.

	n_C			COL_2		PH-`mod`		qPH-`and`		PH-`and`		
name	avg	max	total	avg	max	avg	max	avg	max	avg	max	N
IBM-SF	9.2	30	80860	9.7	30	22.9	128	28.3	560	95.9	8224	8793
JDK1.3.1	4.4	24	32480	4.6	24	8.2	61	10.3	280	51.5	4100	7401
Java1.6	4.4	22	22313	4.7	22	8.3	57	10.7	300	33.8	4097	5074
Orbix	2.8	13	7560	2.8	13	4.4	32	4.1	48	12.3	2050	2716
Corba	3.9	18	6551	3.9	18	6.5	48	6.9	88	19.5	1026	1699
Orbacus	4.5	19	6244	4.7	19	8.1	50	8.8	144	24.0	1026	1379
HotJava	5.1	23	3768	5.5	23	9.8	85	11.6	152	32.8	519	736
JDK.1.0.2	4.6	14	2802	4.7	14	8.4	39	11.1	142	34.2	580	604
Self	30.9	41	55639	31.2	41	66.4	140	79.4	320	181.7	1088	1802
Geode	14.0	50	18442	17.4	50	41.8	267	86.6	816	160.8	1026	1318
Vortex3	7.2	30	14146	7.5	30	15.5	109	20.8	350	79.4	1232	1954
Cecil	6.5	23	6032	6.6	23	12.7	78	18.5	304	53.9	608	932
Dylan	5.5	13	5097	5.5	13	9.0	26	10.2	52	25.5	516	925
Harlequin	6.7	31	4493	7.6	31	13.6	106	17.0	134	39.7	580	666
Lov-obj-ed	8.5	24	3707	11.4	24	19.4	90	22.9	184	53.2	508	436
SmartEiffel	8.6	14	3428	8.6	14	16.3	37	17.0	48	40.0	272	397
Unidraw	4.0	10	2468	4.0	10	6.8	24	7.1	32	17.0	514	614
PRMc	4.5	12	1254	4.5	12	7.5	26	8.6	78	19.5	268	280

loads one thousand classes and interfaces. Overall, implementors might envisage a 16-bit to 32-bit switch.

$\Sigma_C n_C = |\preceq|$ is the size of the transitive closure of the inheritance relation, i.e. a lower bound for all techniques derived from Cohen's test and the ideal linear-space size for variable-size tables. With uniform-size tables, the overall number of entries is $N \max_C n_C$ and encoding class identifiers with bytes instead of short integers does not counterbalance the effect of uniform size (see Variants 2, 3 and 5, pages 11 and 16). The Table then presents the result of class coloring (COL_2), according to the bidirectional variant and global heuristics. The results may vary slightly if different coloring heuristics are used. Finally, perfect hashing (PH) results are presented with modulus and bit-wise `and` hashing functions. These results are produced by exact algorithms, so they would be exactly reproducible were it not for the fact that they depend on some class ordering, which is not uniquely specified. Therefore, all coloring and PH results depend on the test-bed.

— The *coloring* results (COL_2) are very close to n_C, i.e. Cohen's test if inheritance were single. In a framework of global linking or global compilation, the hole number is not significant compared to subobject-based implementations.

— On the opposite side, $N/32$ is the average number of half-words (comparable to all 'avg' columns) required by *direct access* in invariant-reference implementations—it is clearly oversized for the largest benchmarks.

— *Modulus* based perfect hashing is not as good as coloring, but it is almost as good as an efficient-time linear probing. On average, H_C is always smaller than $3n_C$ and slightly exceeds $2\mathrm{COL}_2$. As a good linear probing parameter should be

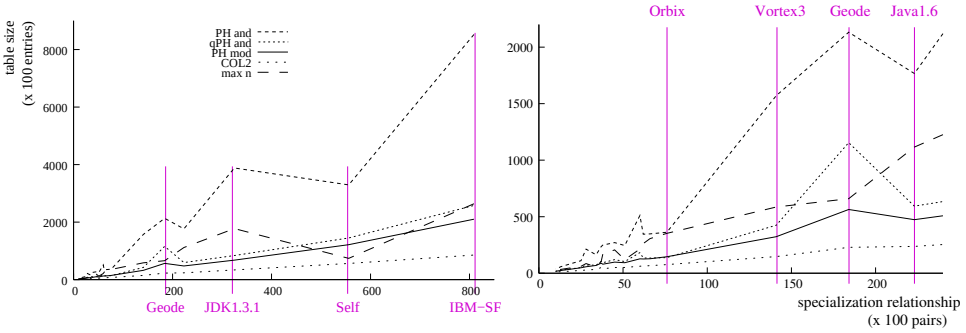


Fig. 12. Near-linearity of perfect hashing on all benchmarks (left) and on all but the largest ones (right). The X-coordinate is the size of the specialization relationship $\preceq$ ('total' column in Table III). The Y-coordinate is the total table size of each technique—they are comparable since all plotted techniques have 2-fold entries. Note that the lines between dots do not have any meaning and it would have made more sense to show scatter plots. However, they would have been barely readable. Curve 'max n' plots the table size for class coloring with uniform-size tables. Perfect hashing sizes include the extra entry for the H_C parameter. Coloring and PH-`mod` are clearly linear, whereas PH- and qPH-`and` are less regular.

close to $2n_C$, the overhead of perfection, i.e. the overhead of perfect hashing w.r.t. linear probing, is not significant.

— With *bit-wise* `and`, the result is not so good—H_C is between 2- and 6-fold greater than with modulus. Actually we have no clear explanation for this difference—a partial argument is that the worst-case value for H_C with `and` is twice that of `mod` (Section 3.2). On the other hand, the H_C average with `and` is still far lower than $N/2$, the average size of direct access tables in subobject-based implementations. Actually, PH improves on direct access (H_C/N) more than it degrades w.r.t. Cohen's test (n_C/H_C).

— *Quasi-perfect* hashing (qPH-`and`) is 2-fold better than PH-`and` for all benchmarks and very close to PH-`mod` on a few of them—on Geode, PH-`mod` remains 2-fold better. In contrast, qPH-`mod`, besides its high cycle count, gave poor spatial results on most benchmarks.

— *Partial perfect hashing* has also been tested—both variants significantly improve PH-`and`, but qPH-`and` remains far better. However, as rootless encoding is free of overhead for static tests, it should be considered.

— Finally, we have also tested *perfect numbering* with bit-wise `and` hashing function but we do not present the results here. On all benchmarks but IBM-SF, the results are markedly better than PH-`and`, sometimes better than qPH or even than PH-`mod`. However, it appears that the technique is highly dependent on the value of the root identifier. In the case of all benchmarks but IBM-SF, the variation does not question these conclusions, but the results for IBM-SF actually run from the best—midterm between PH and qPH—to the very worst—4-fold PH. So, we reserve our conclusions on perfect numbering pending a more in-depth analysis.

Figure 12 shows that the linearity criterion is roughly satisfied—X and Y-coordinates are the respective sizes of the inheritance relation ($|\preceq|$) and of subtype tables.

Table IV. Statistics on method tables: number of introduced and inherited methods, size of method tables with bidirectional coloring (COL_2) and subobject-based multiple inheritance (SMI)—average and maximum per class. The last columns present the ratio of the table sizes for subtyping tests and method invocation in percent, in the case of coloring and for subobject-based implementation, with modulus and bit-wise `and`.

name	methods				method tables				subtype/method (%)			
	introduced		inherited		COL_2		SMI		COL_2	`mod`	`and`	`mod`
	avg	max	avg	max	avg	max	avg	max	COL_2	SMI	SMI	meth
IBM-SF	2.8	257	44.9	346	62.9	397	231.3	2063	8	10	41	26
JDK1.3.1	1.3	149	19.2	243	19.8	243	72.4	1391	12	11	71	21
Java1.6	4.4	286	36.7	669	37.2	669	131.7	3855	6	6	26	11
Orbix	0.4	64	8.3	109	8.4	109	23.0	534	17	19	54	26
Corba	0.4	43	8.0	67	9.9	81	26.9	427	20	24	72	41
Orbacus	1.2	74	18.0	137	18.3	137	68.3	761	13	12	35	23
HotJava	1.8	80	34.2	189	36.0	189	134.8	817	8	7	24	14
JDK.1.0.2	5.3	75	37.0	158	37.1	158	127.3	691	6	7	27	11
Self	14.6	233	577.4	969	586.4	969	3706.2	10098	3	2	5	6
Geode	6.1	193	231.8	880	291.0	892	1445.6	10717	3	3	11	9
Vortex3	0.5	148	156.5	204	156.9	204	1117.7	4994	2	1	7	5
Cecil	2.9	61	78.7	156	80.2	156	441.5	2058	4	3	12	8
Dylan	0.9	64	77.1	139	77.2	139	335.8	1073	4	3	8	6
Harlequin	0.6	62	34.8	129	35.0	129	219.3	977	11	6	18	20
Lov-obj-ed	8.3	117	85.9	289	113.5	289	422.1	1590	5	5	13	11
SmartEiffel	12.2	222	135.3	324	135.4	324	743.5	1576	3	2	5	6
Unidraw	2.9	103	24.1	124	24.1	124	68.9	318	8	10	25	14
PRMc	4.5	107	85.7	173	86.4	174	345.5	1633	3	2	6	4

5.1.2 *Method tables vs. subtype tables.* Table IV presents statistics on methods (introduced or inherited), size of method tables with coloring and subobject-based implementation of multiple inheritance (SMI). In single inheritance, the number of inherited methods gives the exact size of method tables. Table IV ends with ratios between the size of subtype tables and method tables, in coloring and subobject frameworks—the higher the ratio, the higher the overhead entailed by the considered subtype test technique. This ratio is a measure of the relative cost of subtype tests in an existing implementation of basic object-oriented mechanisms. In contrast, the absolute numbers are not so meaningful, e.g. when considering the Self benchmark, whatever the considered technique, absolute numbers in Table III are large but the ratios in Table IV show that subtype tests only require a few percent of method tables.

So this ratio is always less than 25% for coloring and modulus PH, but it may be much lower (down to 1-2%). In contrast, with bit-wise `and`, the ratio is not as low. However, only Java benchmarks have ratios higher than 25%, but the technique would not be used on actual Java benchmarks, and they might be not representative of full multiple inheritance hierarchies (see also Appendix B for a possible explanation). Apart from Java benchmarks, the overhead of perfection—i.e. the ratio $H_C/2n_C$—is not greater than that of subobject-based implementations w.r.t. single inheritance (SMI/COL_2). The last column in Table IV presents the ratio between the hashset size in the `mod` case and the number of inherited methods. This is a weak indication of the expected overhead of perfect hashing in a dynamic typing framework, when each entry takes one half-word, e.g. in Clos or Dylan. However, in these languages with multiple dispatch, there is no natural implementation for

Table V. Classes and interfaces in JAVA benchmarks—total number of classes and interfaces, number of implemented interfaces and extended classes per class (average and maximum per class), and the average entry number of direct access tables, in the straightforward (DA), truncated (tDA) and shared (sDA) variants. The 'total' column presents $\sum_C n_C$, the `implements` relation size. It is also the X-coordinate in the diagrams of Figures 13 and 14.

name	class	interface	implements (n_C)			extends		direct access		
			avg	max	total	avg	max	DA	tDA	sDA
IBM-SF	7920	873	6.2	27	48739	2.8	9	283.2	270.8	209.1
JDK1.3.1	7056	345	1.2	21	8738	3.2	8	39.9	23.0	24.8
yJava1.6	4876	1057	1.6	19	7681	3.1	9	172.6	60.8	137.2
Java1.6	4517	1057	1.7	19	7503	3.0	9	174.2	52.6	147.4
xJava1.6	4804	270	1.5	18	7006	3.0	8	50.6	23.2	35.2
Orbacus	1297	82	1.7	14	2194	2.9	7	10.6	9.0	8.2
Corba	1634	65	1.1	14	1864	2.7	6	6.0	4.6	6.0
HotJava	681	55	2.4	19	1614	2.9	7	10.6	9.2	7.7
Orbix	2676	40	0.3	8	934	2.4	8	1.8	0.9	1.1
JDK.1.0.2	576	28	1.1	9	659	3.6	8	6.4	3.9	2.0

method invocation and it is difficult to conclude either way. If the implementation of method calls relies on subtype testing, as in [Queinnec 1998], there are no method tables at all and a time-efficient test is of course required.

5.2 Application to JAVA

The application of our benchmarks to JAVA implementation is discussed in Appendix B. In particular, xJava1.6 and yJava1.6 are different interpretations of the Java1.6 benchmark. Table V first presents the number of classes and interfaces, along with the average number of entries of the direct access tables (DA) with its truncated (tDA) and shared (sDA) variants. These entries can be bits, bytes, half-words or words according to usage, but method invocation requires half-words for offsets. Truncation and sharing have a significant effect, but they do not change the order of magnitude. We shall no longer consider truncation.

5.2.1 *Hashing of interfaces.* Table VI presents the results of perfect hashing of interfaces, and the respective sizes of the positive and negative parts of the method table. Column 'Pos. offsets' represents what must be implemented at positive offsets, i.e. method addresses[15] and class identifiers for Cohen's test, with 2 identifiers per word. Column 'Neg. offsets' represents what must be implemented at negative offsets, i.e. implemented interfaces—this gives the current n_C parameter. Column 'PH' presents the parameter H_C of perfect hashing tables, i.e. their entry number. As experiments showed that perfect hashing has better results when the identifiers are small, interface identifiers result from a specific numbering of interfaces. So far, all numbers in the table are average per class. Columns 'neg/pos' give the ratio between the corresponding negative and positive sizes. This ratio approximates the relative overhead of each technique. Actually, the exact relative overhead would be $(neg - opt)/(pos + opt)$, whereby *opt* is the size required by the optimal technique, e.g. holeless coloring. However, this optimal is rather small w.r.t. both *neg* and

[15]The method numbers differ from those in Table IV, since the interfaces are removed, but they differ slightly, as they are removed from both the numerator (i.e. table size) and denominator (i.e. class number), and there are only a few of them.

Table VI. Perfect hashing of JAVA interfaces—positive size (extended classes for Cohen's test, and methods), implemented interfaces to be hashed per class (i.e. the current parameter n_C), perfect hashing parameters (PH, with `and` and `mod` functions, and quasi and shared variants, in the `and` case only) and ratio between negative and positive sizes, in percent, for both direct access and perfect hashing in the *offset variant* (average per class). Line 'Total' represents the same ratios as for each benchmark, but computed from the sum of the corresponding parameters on all benchmarks *but IBM-SF*, i.e. when a column depicts some p_b/q_b ratio for each benchmark b, the last line is $\sum_b p_b / \sum_b q_b$.

	pos. offsets			neg. off.	PH				neg/pos (%)				
name	class	meth.	tot.	n_C	`mod`	qPH	sPH	`and`	`mod`	qPH	sPH	`and`	DA
IBM-SF	2.8	44.0	45.9	6.2	13.9	22.2	35.0	40.4	31	50	79	89	310
JDK1.3.1	3.2	19.2	21.3	1.2	2.5	3.2	2.1	4.0	14	18	17	21	96
yJava1.6	3.1	38.5	40.5	1.6	2.7	3.2	2.9	4.7	8	9	11	13	214
Java1.6	3.0	38.3	40.3	1.7	2.7	3.6	3.5	5.8	8	10	12	16	217
xJava1.6	3.0	37.3	39.3	1.5	2.5	3.5	2.7	4.6	8	10	11	13	66
Orbacus	2.9	17.9	19.8	1.7	3.3	3.8	2.8	4.1	19	22	22	23	29
Corba	2.7	7.7	9.6	1.1	2.3	2.9	2.2	3.0	29	36	38	36	37
HotJava	2.9	34.7	36.7	2.4	4.3	5.2	4.1	5.9	13	16	15	18	16
Orbix	2.4	8.2	9.9	0.3	1.3	2.2	0.4	1.4	18	27	19	19	14
JDK.1.0.2	3.6	37.5	39.8	1.1	1.9	2.6	0.6	2.1	6	8	5	6	9
Total	3.0	27.2	29.2	1.4	2.5	3.3	2.5	4.2	10	13	14	16	135

pos. PH is considered here in the *offset variant*, with only one word per PH entry. Remember that the negative space can be multiplied by two if offsets are replaced by addresses, for the gain of one less cycle. The same ratio is also displayed for direct access—here the numbers from Table V represent half-words.

Table VI also presents the ratios between negative and positive parts—the higher the ratio, the higher the overhead. Altogether, one observes the following:

— The relative cost of *Cohen's display* is very low, as there are many more methods than extended classes;

— *Modulus* still gives good results, about $2n_C$, except when there are very few interfaces, as in Orbix—indeed, each hashtable has at least one entry, even when the hashed set is empty. With the offset variant and when excluding IBM-SF, on each benchmark, modulus perfect hashing requires less than 5 words per class—including the half-word for storing H_C—but IBM-SF requires 14 words;

— *Bit-wise* `and` is still inferior to modulus, but acceptable—with the offset variant, on all benchmarks but IBM-SF, it requires only one or two words more than modulus. Furthermore, the PH-`and` average is far lower than the maximum number of implemented interfaces (Table V)—hence, on these benchmarks, PH-`and` is not worse than *uniform-size coloring*. However, IBM-SF requires many more and makes the linearity of PH `and` questionable (Figure 13).

— *Quasi* (qPH) and *shared* (sPH) perfect hashing, both in the `and` case, are midterms between PH-`and` and -`mod`, but only on average and for IBM-SF. They are worse than PH-`and` on some smaller benchmarks. For qPH, this is explained by the fact that a class which implements no interface at all yet requires a non-empty hashtable, with a 2-fold empty entry. For sPH, the reason is the extra pointer in each class. With `mod`, these variants yield even worse results.

— In Table V, the maximum n_C value is far greater than its average, often with a ratio greater than 10—this is a strong argument against *uniform-size* tables; in

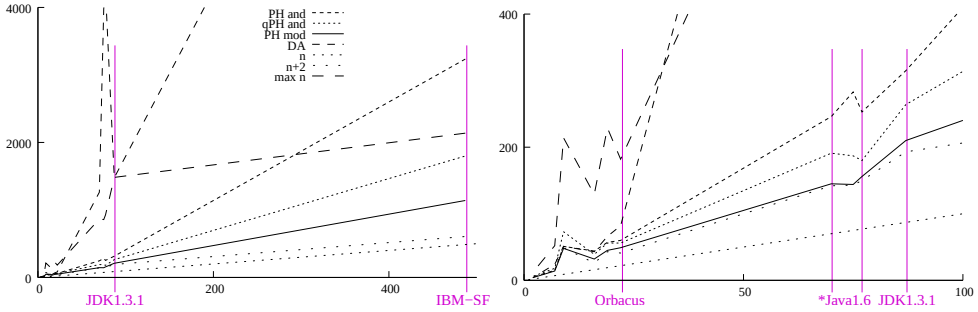


Fig. 13. Near-linearity of interface perfect hashing on all benchmarks (left) and all but IBM-SF (right). X-coordinate is the size of the `implements` relationship (column 'total' in Table V), i.e. the restriction of $\prec$ between classes and interfaces. PH-`mod` linearity is very good, but IBM-SF makes PH-`and` less linear. Direct access (DA) proves its bad scalability. The 'n' line, which has equation $x = y$, plots the n_C parameter and is the ideal lower bound, i.e. global coloring without holes; the 'n+2' line, i.e. $n_C + 2$ entries for each class C, is a lower bound for techniques based on searching and caching or for incremental coloring with bound checks (IC-`bc`). Line '$\max n$' plots a lower bound of uniform-size class coloring (IC-`us`). All Y-data are comparable and based on both short integer ID encoding and the offset variant—all techniques but 'DA' have 2-fold entries, so the data for 'DA' has been divided by 2.

practice, it means that the technique proposed by Palacz and Vitek [2003] would require 6 words per class in the HotJava benchmarks, only for the subtyping test, whereas PH includes method invocation for the same price;

— Finally, *direct access* confirms its bad scalability: its results are similar to PH on all small benchmarks, but truly oversized with IBM-SF, JDK1.3.1 and the variants of Java1.6. As there is some evidence—see Appendix B—that our benchmarks underestimate interface numbers, DA is actually ruled out, even with sharing.

To assess the scalability of the different techniques, Figure 13 plots, for all benchmarks, the total table size as a function of the size of the `implements` relationship. Figure 14 separately plots, for each of the largest benchmarks, both coordinates as classes are successively loaded. Figure 15 plots PH-`and` for all benchmarks.

— Despite the apparent step-wise behavior, due to the fact that a class implementing no interface at all still requires one entry, there is some evidence of space-linearity with perfect hashing. In Figure 15, the curves of all benchmarks are clearly very close to the prefix of that of IBM-SF.

— *Incremental coloring* is lower bounded by the '$\max n$' curve in the uniform-size variant (IC-`us`), and by the '$n + 2$' curve in the variant with bound checks (IC-`bc`). These are lower bounds, hence, according to heuristics, actual values might be 50% higher. This clearly shows that IC-`us` is oversized on all benchmarks but IBM-SF, whereas IC-`bc` is not better than PH-`mod`, except for IBM-SF.

— Sharing has a poor effect on PH—due to the required extra pointer, it just flattens the curve—but it markedly improves IC, because there is no extra pointer.

5.2.2 *Hashing of methods and interfaces.* We have also tested hashing of methods and interfaces (Section 4.2.2). As expected, the approach turns out to be

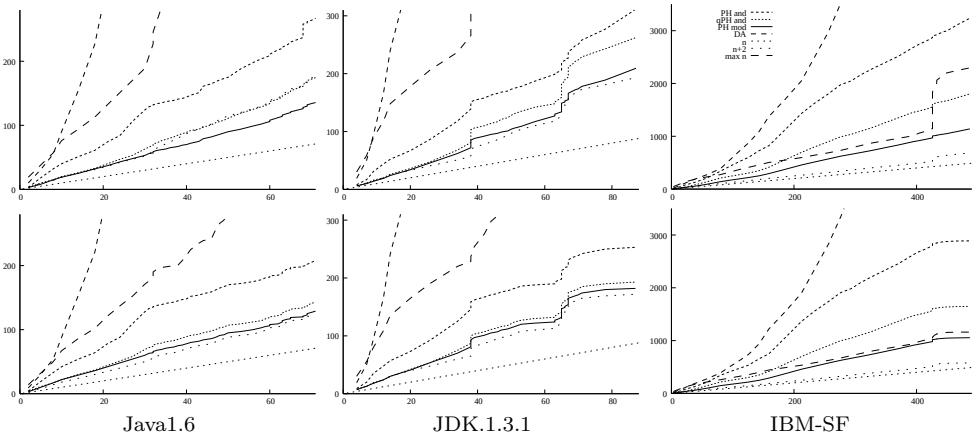


Fig. 14. Scalability of the different techniques for the largest benchmarks, without (top) and with sharing (bottom). The curves plot the same X- and Y-coordinates as in Figure 13, as classes are successively loaded. In the '$\max n$' curve, the maximum is taken from the whole considered hierarchy—hence, this accounts for fixed-size tables—but there is very little difference when considering uniform-size resizable tables.

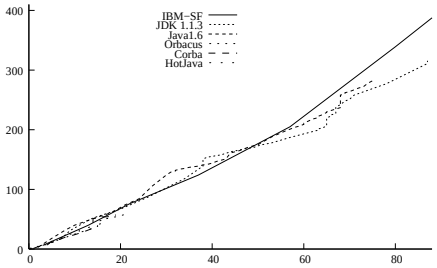


Fig. 15. Scalability of PH-`and` on all benchmarks—all curves are roughly aligned, showing that all benchmarks behave in a similar way.

unrealistic—even with modulus and in small benchmarks, the negative part can be larger than the positive one. With bit-wise `and` and larger benchmarks, the ratio exceeds 10.

5.2.3 *Conclusions for* Java. Several versions of perfect hashing involved comparing, according to the particular variant (PH, sPH, qPH), what is hashed (interfaces or methods), and which hashing function is used (`and`, `mod` or even the hypothetical 2-parameter function—2PH). Direct access to interfaces was also considered, at least in its shared variant (sDA). Moreover, in the case of interfaces (hashing or direct access), one may also distinguish between addresses and offsets, with the latter dividing the size by two but adding one cycle and one instruction. Incremental class coloring (IC) must also be considered, in its improved form with variable-size tables and bound checks (IC-`bc`), and with uniform-size tables (IC-`us`), for comparison. However, we did not simulate space—we only consider the '$n + 2$' and '$\max n$' lower bounds (Figures 13 and 14).

Direct access and method perfect hashing have been ruled out, as they appear to be truly oversized. Table VII lists the remaining techniques, ordered by the cycle count of subtype test and method invocation, and presents cycle counts and sequence lengths for both mechanisms, together with an estimation of the mem-

Table VII. Comparison of the remaining techniques for JAVA, lexicographically ordered by cycle counts for subtype tests, then method invocation (with $L = 3$, $B = 10$ and $D = 6$). The first two rows are the single subtyping reference. In the cycle columns, all lines but the first present the difference with the first line, function of L, then the total number of cycles and the possible number of extra cache miss or misprediction risks. Column 'neg/pos' gives the ratio between positive and negative sizes, in percent. The last column is twofold and presents the space ratio for IBM-SF alone (left) and for all benchmarks but IBM-SF (right).

technique		method call				subtype test				space (%)	
		cycles		risk	code	cycles		risk	code	neg/pos	
SST	Cohen	$2L + B$	16		3	$2L + 2$	8		4	–	–
SST	Schubert	–	–		3	$+2$	10	1	8	–	–
IC-`us`	offset	$+2L + 2$	24	2	8	$+L + 1$	12	2	7	> 63	68
IC-`bc`	**offset**	$+2L + 2$	**24**	**2**	**8**	$+L + 2$	**13**	**3**	**10**	> 19	13
PH-`and`	address	$+2L + 2$	24		7	$+L + 3$	14		8	180	32
PH-`and`	**offset**	$+2L + 3$	**25**		**8**	$+L + 3$	**14**		**8**	**89**	**16**
sPH-`and`	offset	$+2L + 3$	25	1	9	$+L + 3$	14	1	9	79	14
2PH-`??`	offset	$+2L + 4$	26		10	$+L + 4$	15		10	??	??
qPH-`and`	offset	$+2L + 5$	27	1	12	$+L + 4$	15	1	11	50	13
PH-`mod`	address	$+2L + D + 1$	29		7	$+L + D + 2$	19		8	62	20
PH-`mod`	**offset**	$+2L + D + 2$	**30**		**8**	$+L + D + 2$	**19**		**8**	**31**	**10**

ory usage, given by the ration 'neg/pos' from Table VI. SST techniques have been added for comparison. Note that, regarding space, by no means do shorter code sequences compensate for larger tables—the number of sites of interface-typed method invocations is rather low, certainly not many more than a few per class[16].

— Interface PH with addresses has very little support and PH-`and` with offsets should be preferred in all cases—vs. PH-`mod` with addresses, which is hardly more compact, or vs. PH-`and` with addresses which remains unreasonably oversized for IBM-SF, with very little gain.

— Interface PH-`and` is the preferred midway solution while programs do not reach IBM-SF size.

— On the opposite side, interface PH-`mod` with offsets is the solution if space is the priority, as the space increase is about 10% for smaller benchmarks, or if the target program approaches the IBM-SF size.

— sPH and qPH might also be considered, but their improvement upon PH-`and` or -`mod` is not significant.

— Incremental coloring must also be considered, despite its hazardous behaviour at load-time—IC-`bc` must be preferred to IC-`us`, as its speed is close to that of PH-`and`, and its memory usage lower bound on small benchmarks is a midterm between PH-`mod` and PH-`and`. This lower bound is not as low as one might expect because of the 2 extra pointers which are required for memory management.

When disregarding IBM-SF, PH-`and` is certainly the best tradeoff between runtime and load-time efficiency. For IBM-SF-like programs, PH-`mod` or IC-`bc` should

[16]Ducournau [2002a] reports an average of less than 10 method definitions per type and one of the few available statistics on the number of call sites, Driesen et al. [1995], indicates an average of 4 call sites per method definition. For instance, the single instruction saved on by method perfect hashing cannot compensate for the 58 extra words per class entailed by bit-wise `and` with IBM-SF.

be considered, but the latter must prove its scalability from the space—we only consider, here, lower bounds—and load-time standpoints.

A last concern is modulus latency, which depends on processors, so the conclusion must take processor characteristics into account. However, the present conclusions are somewhat independent of a precise assessment of the integer division latency—indeed, our optimistic 6-cycle value places modulus PH at the very end of the file, from a cycle standpoint. Longer latencies would only increase the difference.

Altogether, the PH variants are very close to each other in terms of load-time computation, generated code and data structure, and the choice depends on the context—programming style, program size or even processor version. Hence, they could be a parameter of the run-time system. For instance, SCALA programs would run with a `nosharing` option and very large programs would use a `space` priority.

6. RELATED WORKS

Many people have contributed to subtyping tests.

Mathematical encoding of posets. This paper focuses on techniques inherited from Cohen [1991], as they are efficient in practice. A rather different approach is based on mathematical encoding of partial orders [Aït-Kaci et al. 1989; Caseau 1993; Habib and Nourine 1994; Habib et al. 1995; Habib et al. 1997; Raynaud and Thierry 2001]. The objective is then optimal encoding, thus minimizing the size. Fall [1998] surveyed this approach, which is based on mathematical properties of posets like dimensions or products. However, a minimal size may be unrealistic, in practice, as the code for extracting the data becomes quite intricate. A good example is given by *bit pack encoding* [Vitek et al. 1997] which is a variant of *pack encoding*, i.e. class coloring, where the class identifiers are no longer global but depend on the class color and the size of the field associated with a color depends on the number of classes with the same color. The resulting tables are markedly smaller, but the decoding code is markedly longer. Moreover, these optimized encodings are incompatible with dynamic loading, as they are not incremental and their complexity is too high, i.e. often non-polynomial.

Combination of coloring and Schubert's numbering. Cohen's display and Schubert's numbering are the best techniques from the time standpoint and they differ by the encoding direction. A class encodes its superclasses in the former, its subclasses in the latter. Obviously, a bottom-up encoding is in favour of incrementality. Both techniques have been generalized to multiple inheritance, the former with coloring and the latter with *range compression* [Agrawal et al. 1989], where several intervals can be associated with each class. The test is now not more incremental, and it is no longer time-constant. Several authors propose to combine both approaches in such a way that each class is associated with (1) a *color* (equivalently, the set of classes is partitioned in *slices*), (2) one or more identifiers, (3) one or more intervals. With *PQ-encoding* [Gil and Zibin 2005], each class is associated with one identifier in each slice and a single interval for encoding its subtypes. It gives one of the most compact constant-time techniques available for multiple inheritance, but it is not incremental and the encoding is too complex to recompute it at each class loading. In contrast, Alavi et al. [2008] propose a similar though inverted combination for encoding the supertypes. Each class is now associated with a single identifier and

one interval in each slice. However, with a single interval in each slice, the technique is not more incremental and the authors avoid global recomputations by allowing several intervals per slice. This closely resembles *range compression*, apart from the encoding direction. Hence, the test is no longer time-constant.

Incremental coloring (IC). In the framework of memory-constrained real-time systems in JAVA, Palacz and Vitek [2003] propose another combination of these two non-incremental techniques. Schubert's numbering (Variant 4, page 12) is used for class targets, whereas class coloring (Variant 6, page 16) is used for interface targets. Of course, class and interface targets are independent and one could also combine Cohen's test for class targets with coloring for interface targets, or Schubert's numbering for classes with perfect hashing for interfaces. So, apart from method invocation— the paper does not cover this aspect—the proposal of Palacz and Vitek is functionally very close to ours and the run-time efficiencies are comparable. Regarding Schubert's numbering, the authors propose two alternatives, namely lazy or eager recomputation. As lazy recomputation obviously makes the test time-variable, they finally chose eager recomputation for real-time frameworks. From the space standpoint, Schubert's numbering is theoretically unbeatable, but the average size of Cohen's display is almost never greater than 2 (column 'extends' in Table V)—so the gain is low.

Regarding coloring, they use the uniform-size (IC-`us`), byte-encoding variant (Variant 3, page 16). As byte-encoding does not counterbalance the overhead—moreover, it would not apply with method invocation—the original proposition compares badly with perfect hashing from the space standpoint. We have discussed (in Section 2.2.2) the possible solutions and statistics show that the most compact implementation is based on variable-size tables, allocated as usual objects—so, explicit bound checks cannot be avoided (IC-`bc`). Figures 13 and 14 show that the resulting technique, lower-bounded by the '$n + 2$' curve, is not much better than PH-`mod`, but significantly better than PH-`and`. Furthermore, color tables could be shared without any overhead (Section 4.3.3)—but the paper does not mention it. From the time standpoint, our evaluation shows that incremental coloring and PH-`and` are very close, but incremental coloring adds 2 cache miss risks in both mechanisms, subtype testing and method invocation.

The main drawback of coloring is at load-time—non-incrementality implies recomputations and reallocations, and all of this is worsened by multi-threading. In the worst case, perfect hashing should be markedly better than incremental near-optimal coloring from the recomputation standpoint. PH-`and` does not entail any recomputation and has a $\mathcal{O}(n_C \log id_C)$ complexity—where n_C is the number of superclasses of the loaded class C and id_C is its class ID, i.e. the number of currently loaded classes (Appendix A.3). In contrast, incremental coloring may entail full recomputation. Optimal coloring is NP-hard and the heuristics presented in [Ducournau 2006] have $\mathcal{O}(N^k)$ complexity, where N is the total number of classes and k some small constant, not less than 3. In contrast, the heuristic presented by Palacz and Vitek seems to have $\mathcal{O}(n_C id_C)$ or $\mathcal{O}(id_C^2)$ complexity per class[17], which

[17]This is the complexity in full multiple inheritance. With multiple subtyping, $\mathcal{O}(id_C^2)$ becomes $\mathcal{O}(N_c N_i)$, where N_c (resp. N_i) is the number of currently loaded classes (resp. interfaces).

is consistent with the $\mathcal{O}(N^3)$ global complexity. However, as aforementioned, this heuristic favors time to the detriment of space and an exact assessment of its space and time efficiency should be made—the paper actually compares the proposed technique only with the naive technique of the underlying virtual machine. Furthermore, their benchmarks are not directly comparable to ours, since they consider real programs, with relatively few loaded classes, whereas we consider complete class hierarchies. There is an order of magnitude between class numbers in both tests. Their numbers are more representative of actual programs than ours but they fail to prove the scalability of the approach—though this was likely not the authors' intention, since they place themselves in a memory-constrained real-time setting, likely with small programs. Overall, the run-time efficiency of incremental coloring can be seen as a midterm between PH-`and` and PH-`mod`—the speed of the former and the compactness of the latter. Nevertheless, load-time efficiency and simplicity are in favor of perfect hashing.

Incremental coloring could also be applied to full multiple inheritance in invariant-reference implementations, though this would exacerbate its drawbacks, but we are not aware of such experiments.

Subtype testing in Java. Cohen's test is commonly used in the implementation of Java virtual machines for class subtyping tests [Alpern et al. 2001b; Click and Rose 2002]. However, it is often used in a non-optimal way, with uniform-size separate tables (Variant 2, page 11). Regarding interface subtyping tests, Alpern et al. [2001b] use a direct access array indexed by interface identifiers. However, the technique does not seem to be used for method invocation. Therefore, tables are smaller but another unknown data structure is used for it. Some other experimental implementations, e.g. [Krall and Grafl 1997] and [Gagnon and Hendren 2001], use direct access—our simulation shows that even direct access to interfaces is over space-consuming. Direct access to methods is worse by an order of magnitude. So, Gagnon and Hendren [2001] allocate objects in the empty areas of these huge tables. This is an interesting idea, but we have no information on how much space is saved in this way.

Searching and caching. In contrast with these rather sophisticated approaches, many production virtual machines use naive techniques. For instance, according to [Palacz and Vitek 2003], the Sun Microsystems Research VM uses a simple linear search improved with a cache in the source type memorizing the last target type for which the test succeeded. The technique of Click and Rose [2002], which is considered to be very efficient, improves this by combining it with Cohen's test. Of course this cache improvement might serve for any table-based subtyping technique. In the best cases, the cache yields a 2-`load` test, like Cohen's test, instead of 3-`load` with perfect hashing[18]. However, cache misses would entail, in the PH case, 2 extra `load`s, 2 `store`s, plus an extra test and its associated misprediction risk. Obviously, the improvement is a matter of statistics and those presented in Palacz and Vitek [2003] show that, according to the different benchmarks, cache miss rates can be as low as 0.1% or more than 50%. Regarding space, Click and Rose [2002] report 8–11 words per class. This is larger than modulus interface perfect hashing, except for

[18]Contrary to some authors, we include here the access to the method table from the object itself.

IBM-SF which seems to be larger than the hierarchies tested by Click and Rose—indeed, they report a maximal depth of always less than 8, whereas IBM-SF and Java1.6 reach 9 (column 'extends' in Table V). Moreover, linear search coupled with cache requires $n_C + 2$ entries for each class C. As the average n_C is low—it exceeds 2 only on two benchmarks—adding two entries yields significant relative overhead. Figures 13 and 14 show that PH-`mod` compares favourably on all benchmarks but IBM-SF. Finally, they do not address method invocation.

JAVA *`invokeinterface`.* Alpern et al. [2001a] review interface typed method invocation in JAVA and .NET languages. As for subtype testing, a basic approach is linear search. In Jalapeño, Alpern et al. [2001a] associate a fixed-size hashtable with each class, mapping each method identifier to the address of a method or of a decision tree indexed on method identifiers. This could be considered as a perfect hashtable—but the value associated with an entry may be a dispatch tree—or a classic hashtable using separate chaining, where the collision resolution is no longer interpreted, but compiled instead. The advantage is that hashing is uniformly and statically done—no `load` or hash function at run-time and a compiled collision resolution chaining is likely more efficient than interpreted linear probing. The drawback is inconstant time—though the `load` number is constant and small—together with the fact that it does not work for subtyping tests. This paper presents benchmarks with two values, 5 and 40, for the H parameter. The former is in the same region as the average H_C in PH-`mod` and PH-`and` in all benchmarks but IBM-SF (Table VI)—i.e. with the same space cost, this technique replaces dynamic hashing by compiled probes. The latter is the average H_C in PH-`and` for IBM-SF—hence, it is much more costly than PH-`and` for all other benchmarks.

Bidirectionality. Bidirectionality of method tables has become quite common in virtual machines, at least for experimental ones, e.g. Krall and Grafl [1997] and Gagnon and Hendren [2001], and derives from Pugh and Weddell [1990] and Myers [1995]. It markedly improves coloring [Ducournau 2006] but might, however, be a problem for reflexive implementations—e.g. JAVA implementations of the JAVA virtual machine—since bidirectional object layouts are much more uncommon. Hence bidirectionality is likely not desirable in the context of incremental coloring for JAVA.

Binary tree dispatch (BTD). On the other hand, not all object-oriented implementations are based on method tables. In SMART EIFFEL, the GNU EIFFEL compiler, method tables are not used. Instead, objects are tagged by their class identifier and all polymorphic operations—particularly subtyping tests—are implemented using small dispatch trees [Zendra et al. 1997; Collin et al. 1997]. However, the approach is practical only because compilation is global, hence all classes are statically known. Furthermore, type analysis restricts the set of *concrete types* [Bacon and Sweeney 1996; Grove and Chambers 2001] and makes dispatch efficient. BTD is also an interesting example of the possible disconnection between code length, hence inlining, and time efficiency. Indeed, here both values are in an exponential relationship—hence proving that not all efficient code sequences are inlinable.

In a dynamic typing framework, Queinnec [1998] proposed a method dispatch technique based on a variant of BTD, whereby each test is a subtype test instead

of an equality test. The proposal was limited to single inheritance.

Other usages of perfect hashing. In the programming language area, perfect hashing has been used for closed symbol tables, e.g. reserved words [Schmidt 1990], or for constant-time access to sparse arrays. There is little evidence of its usage in an object-oriented framework, especially when applied to pure object-oriented mechanisms. Driesen [1993a], Zibin and Gil [2003a] discard it for method dispatch, as they consider that it is too complicated, with a too high constant or complexity—actually, they consider *minimal* perfect hashing functions, i.e. functions such that $H_C = n_C$. Klefstad et al. [2002] use perfect hashing for method dispatch in the framework of CORBA distributed applications—method names are hashed, which is more expensive than hashing small integers. *Sparse table compression* can be considered as a special case of perfect hashing [Czech et al. 1997]. Besides coloring, this includes *row displacement* [Tarjan and Yao 1979] which has been applied to method invocation by Driesen [1993b; 2001], Driesen and Hölzle [1995]. Row displacement has not been explicitly proposed for subtype tests, though it would obviously apply—following the analogy between method call and subtype test underlying Cohen's test (Section 2.1.2, page 9). However, row displacement is compatible with dynamic loading only in its *class-based* variant—according to Driesen, this might be less space-efficient than the selector-based variant.

Finally, to our knowledge, the various perfect hashing functions proposed in the literature do not meet our requirements. For instance, *quotient reduction* [Sprugnoli 1977] involves a function $x \mapsto \lfloor (x + s)/n \rfloor$, which presents the same disadvantage as `mod` augmented by an extra parameter, plus a need for a bound check and very poor compactness with consecutive numbers.

7. CONCLUSION AND PERSPECTIVES

In this paper, we have proposed a subtype testing technique based on perfect hashing, a truly constant-time variant of hashtables. This generalizes the test proposed by Cohen [1991] and class coloring [Vitek et al. 1997] to both multiple inheritance and dynamic loading. Indeed, class coloring is a special case of perfect hashing, where all hash functions are the coloring function, i.e. $\forall C \preceq D, h_C(D) = \chi(D)$. Of course, this similarity does not account for one major difference, i.e. coloring has an extensional definition whereas all hashing functions are explicit functions.

The proposed technique has been considered in different frameworks, namely an abstract invariant-reference implementation of multiple inheritance, C++ subobject-based implementation and JAVA multiple subtyping, for interfaces only. In the latter case, the technique also applies to method invocation, in a partial converse of the analogy between method call and subtype testing (page 9). Perfect hashing and its different variants—with modulus or bit-wise `and`, quasi-perfect hashing—has been compared to some known constant-time techniques, class coloring and direct access. An additional contribution of this paper is that these alternative techniques themselves have been presented in improved and partially novel forms. Cohen's display and class coloring benefits from being allocated in dedicated memory areas—both bound checks and uniform size are avoided. In the multiple subtyping framework, sharing of interface tables significantly improves direct access and incremental coloring. Furthermore, they all also apply to method invocation

in a JAVA framework. Incidentally, a minor contribution of this paper is that it highlights that both mechanisms resort to common implementation, especially in the context of JAVA interfaces—we actually did not find any paper addressing both mechanisms at the same time.

7.1 Evaluation

The comparison between the considered techniques relies on the estimation of cycle count and code length, based on an abstract assembly language proposed by [Driesen 2001], together with a simulation of the size of all involved data structures on a large set of real-size benchmarks. Altogether, the resulting subtype test fulfills our initial requirements.

Constant-time. In an invariant-reference implementation, the test with bit-wise `and` takes $3L + 5$ cycles, almost the same as incremental coloring and shared direct access, but without any extra cache miss risk; modulus would add the latency of integer division; subobject-based implementations add uniform overhead to all techniques.

Linear-space. Our empirical results show that perfect hashing is space-linear with modulus, but this is less clear for bit-wise `and` because of the largest benchmark, IBM-SF. With modulus, the number of entries of tables dedicated to subtyping tests is about twice that of global bidirectional coloring; with bit-wise `and`, the ratio is not as good but quasi-perfect hashing is close to PH-`mod`. In a subobject-based implementation, the size doubles but the method tables also present ratios of the same order—this is the price to pay for an efficient-time subobject-based implementation.

Multiple inheritance. Perfect hashing works well with plain multiple inheritance—using hashsets or hashtables, according to implementations—and it may apply to JAVA in a more efficient way, i.e. restricted to interfaces, and in a more general mechanism, applied also to method invocation.

Dynamic loading. The technique is inherently incremental—at load or link time, it requires only class numbering together with the computation of H_C with straightforward low-complexity algorithms and the definitive allocation of the corresponding data structure; in contrast, incremental coloring might entail bad-case recomputations of higher complexity, with costly memory management.

Inlining. The code sequence takes 8 instructions in an invariant-reference implementation, i.e. almost like incremental coloring.

Altogether, besides JAVA-like languages, perfect hashing can be recommended for all languages which rely on both multiple inheritance and dynamic loading (or linking)—CLOS, DYLAN, PYTHON or C++, the latter despite the inherent overhead of subobjects.

Clearly, these results are not perfect, i.e. smaller, shorter, faster would be better, and the comparison with alternative techniques is not straightforward. In this paper, efficiency assessment is mainly abstract, as it implies an abstract computational model and abstract class libraries. In this framework, no technique surpasses all others on all criteria and perfect hashing compares well with other techniques. From the time standpoint and the code sequence length, PH-`and`, incremental col-

Table VIII. Final comparison on the four main criteria—each technique has its dark side

	time	code	space	load
PH-`and`	+	+	−	+
PH-`mod`	−	+	+	+
IC-`bc`	+	+	+	−

oring and shared direct access are evenly matched. The slight difference concerning PH is counter-balanced by the extra cache miss risks in the other techniques. From a purely temporal standpoint, PH-`mod` compares badly, due to the high latency of integer division. From the space standpoint, direct access must be ruled out and the other results depend on whether inheritance or subtyping are multiple. In multiple subtyping, PH-`mod`, PH-`and` and IC-`bc` are almost evenly matched on all benchmarks, but IBM-SF which highly degrades PH-`and`. In contrast, in full multiple inheritance, PH-`mod` is markedly better than PH-`and` and there is no empirical evidence of the scalability of IC—there is currently no alternative to PH that satisfies all five criteria. Finally, from the load-time standpoint, PH and direct access are inherently incremental and only need very simple computations (Appendix A) and memory allocation. On the contrary, incremental coloring requires partial recomputations which may be costly and can entail reallocation of a whole set of class ID tables. Hence, these tables must be allocated in fully dynamic memory and their run-time management may be costly. Multi-threading worsens the situation. Another element makes the comparison difficult. Palacz and Vitek [2003] tested incremental coloring by running a set of real programs, made of 10-thousand class libraries but loading only 1-thousand classes. In contrast, our tests are more pessimistic as they simulated the total loading of similar class libraries. So, the scalability of PH has actually been assessed on very large benchmarks—with imperfect results on very large hierarchies and some doubt on the representativeness of the largest benchmark—whereas the scalability of incremental coloring on several thousand class loadings remains a conjecture.

So, PH strong points are its inherent incrementality, which entails no recomputation at load-time and no extra cache miss risk at run-time, together with its short and efficient code sequence. Its main weakness is that the considered hash functions lead to a time-space tradeoff. However, in the present state of affairs, this is a practical technique that deserves a more in-depth evaluation in production run-time systems.

7.2 Perfect hashing in practice

Besides the abstract assessment presented in this article, realistic tests should involve implementing perfect hashing in run-time platforms and measuring the run-time parameters of some real programs. This was clearly beyond the scope of the present paper. The approach might be applied to C++ compilers, under two minor technical conditions: (i) linkers should be adapted to class numbering and H_C computation; (ii) perfect hashing should be adapted to *non-virtual inheritance*, i.e. an impure form of multiple inheritance used in this language when superclasses are not annotated by the `virtual` keyword [Ellis and Stroustrup 1990; Lippman 1996; Ducournau 2002a]. The application to CLOS, DYLAN, JAVA and .NET languages is more straightforward as their runtime systems are equipped with a loader. For

JAVA and .NET languages, perfect hashing is a complete single interface implementation solution for subtyping test and method invocation. In the particular framework of CLOS or DYLAN *generic functions*, perfect hashing could also serve as a basis for method dispatch, in a way similar to the proposal of Queinnec [1998]. However, we have not considered the case of unloading and reloading classes in our requirements. This should not be a problem, at least as long as unloading a class entails unloading all its subclasses.

Choosing between the two hashing functions involves a time-space tradeoff. In the framework of subobject-based implementation, the overall size of subtyping test tables is substantially smaller than method tables in the modulus case. Moreover, in C++, the subtyping test is generally not critical and current implementations, e.g. `g++`, are markedly slow—see for instance the benchmarks in [Privat and Ducournau 2005]. Therefore, modulus perfect hashing is a good tradeoff. In contrast, application to method invocation—in JAVA or, hypothetically, in CLOS—should favor time. Moreover, subtyping checks are quite common in JAVA programs, due to the lack of genericity (up to version 1.5) and to covariant array subtyping. Among the various techniques that we have tested in the JAVA framework, only a few are finally preferred—perfect hashing of interfaces, with `mod` or `and` functions, in both cases with the offset variant, and incremental coloring with explicit bound checks. Load-time considerations should exclude coloring, though run-time efficiency makes it a good tradeoff between PH-`mod` and PH-`and`, even on the largest benchmarks. Since it becomes degraded only on a very specific JAVA benchmark, PH-`and` may be recommended as long as programs have only a few thousands classes and interfaces, which should be the common situation for some time. Finally, in the near future, program size will go on increasing but programming style will also evolve. Our JAVA benchmarks are likely representative of a certain programming style, where the use of interfaces is limited because programmers are not fully aware of interfaces, or they suspect their inefficiency[19]. Future benchmarks might be more demanding. Furthermore, when JAVA interfaces are not hand-made but generated by re-engineering [Huchard and Leblanc 2000] or by the compiler, say of the SCALA language [Odersky et al. 2008], the usage of JAVA interfaces can be completely novel and the choice different. This stresses the need for making interface implementation an open parameter of run-time systems but this also questions the choice of the JAVA run-time system for implementing a language such as SCALA.

7.3 Perspectives

Anyway, it would certainly be worthwhile to search for more efficient hashing functions—we could not imagine another function than modulus and bit-wise `and`. In the JAVA framework, quasi-perfect hashing does not sufficiently improve the size to counter-balance a 2-probe test—it is however markedly better for full multiple inheritance in a reference-invariant implementation. A better solution might be a two-parameter function family, which would add 1 cycle and 2 instructions to bit-wise `and`, and should reduce the space to that of modulus, or even less. However, this 2-parameter function still has to be found. Among the different variants that we have described, *perfect numbering* is surely the most appealing, since it

[19]Consider, for instance, [Alpern et al. 2001a] subtitle: *`invokeinterface` considered harmless.*

has the run-time efficiency of PH-`and`, together with a good expected compactness. However, preliminary tests revealed an erratic behavior in the specific case of the IBM-SF benchmark. So, further studies are required to gain insight into the reasons for this strange behavior and to find a way to overcome it. Furthermore, we have defined perfect numbering as the optimization of the identifier of a single class. However, in practice, when the runtime system triggers the loading of class C, it must also load all the superclasses of C which have not yet been loaded. Therefore, the optimization should not concern a single class but a whole set of classes. This should also be the right way to apply perfect numbering to JAVA interfaces. Of course, this requires a precise formulation of the optimization problem, algorithms or heuristics for computing optimized identifiers, and a class-loading model for running simulations and assessing the technique.

Another perspective is more theoretical. So far, our assessment of PH-`mod` and PH-`and` has been purely empirical. However, an in-depth mathematical analysis might help to determine the average behavior of these techniques. The problem is similar to previous studies on hashing, e.g. [Knuth 1973; Vitter and Flajolet 1990], except that subtype testing does not involve a single hashtable, but a set of dependent hashtables. Therefore, the exact problem must be stated as follows. Given some distribution of posets $(X, \preceq)$, with an associated one-to-one numbering $id : X \to [0..|X| - 1]$ that verifies $x \prec y \Rightarrow id(x) > id(y)$, what is the statistical behavior of $\sum_{C \in X} H_C$, where H_C is the optimal hash parameter associated with the considered perfect hashing function?

Perfect hashing could also be applied to object layout, as a kind of *inherently incremental coloring*. In this case, reference invariant holds and the object layout is the same as with single inheritance, except that the position invariant does not hold. Instead, attributes and methods are both grouped according to the class which introduces them. The method table negative part is the hashtable with 3-fold entries—i.e. class identifier, offsets of the corresponding attribute group in the object layout[20] and method group in the method table positive part. Overall, subtype testing and method invocation would be the same as previously described for JAVA interfaces. On the other hand, accesses to attributes would require two indirections—this is considerable compared to single inheritance but not much worse than with subobject-based implementation, when the receiver's static type is not the class introducing the attribute. This would, however, apply only in a static typing framework.

On the whole, perfect hashing is a 30-year old technique that we propose to apply to a 25-year old problem. We were surprised to conclude, following an in-depth bibliographical search, that this had not yet been done.

ACKNOWLEDGMENTS

This work has been partially supported by grants from Région Languedoc-Roussillon (034750). The author thanks the anonymous referees for their helpful comments and various people in LIRMM for fruitful discussions on hashing, processors and integer division.

[20]This technique is called *field dispatching* in Zibin and Gil [2003b] and *accessor simulation* in Ducournau [2002a; 2006].

REFERENCES

Agrawal, R., Borgida, A., and Jagadish, H. 1989. Efficient management of transitive relationships in large data and knowledge bases. In *Proc. SIGMOD'89*. ACM SIGMOD Record, 18(2). 253–262.

Aït-Kaci, H., Boyer, R., Lincoln, P., and Nasr, R. 1989. Efficient implementation of lattice operations. *ACM Trans. Program. Lang. Syst. 11,* 1, 115–146.

Alavi, H. S., Gilbert, S., and Guerraoui, R. 2008. Extensible encoding of type hierarchies. In *Proc. POPL'08*. ACM, 349–358.

Alpern, B., Cocchi, A., Fink, S., and Grove, D. 2001a. Efficient implementation of Java interfaces: Invokeinterface considered harmless. In *Proc. OOPSLA'01*. SIGPLAN Notices, 36(10). ACM Press, 108–124.

Alpern, B., Cocchi, A., and Grove, D. 2001b. Dynamic type checking in Jalapeño. In *Proc. USENIX JVM'01*.

Bacon, D. and Sweeney, P. 1996. Fast static analysis of C++ virtual function calls. In *Proc. OOPSLA'96*. SIGPLAN Notices, 31(10). ACM Press, 324–341.

Barnes, J. 1995. *Programming In Ada 95, first edition*. Addison-Wesley.

Cardelli, L., Ed. 2003. *Proceedings of the 17th European Conference on Object-Oriented Programming, ECOOP'2003*. LNCS 2743. Springer.

Caseau, Y. 1993. Efficient handling of multiple inheritance hierarchies. See OOPSLA [1993], 271–287.

Chaitin, G. J. 1982. Register allocation & spilling via graph coloring. In *Proc. 1982 ACM SIGPLAN Symposium on Compiler Construction*. 98–105.

Click, C. and Rose, J. 2002. Fast subtype checking in the Hotspot JVM. In *Proc. ACM-ISCOPE conference on Java Grande (JGI'02)*. 96–107.

Cohen, N. 1991. Type-extension type tests can be performed in constant time. *ACM Trans. Program. Lang. Syst. 13,* 4, 626–629.

Collin, S., Colnet, D., and Zendra, O. 1997. Type inference for late binding. the SmallEiffel compiler. In *Proc. Joint Modular Languages Conference*. LNCS 1204. Springer, 67–81.

Czech, Z. J. 1998. Quasi-perfect hashing. *The Computer Journal 41*, 416–421.

Czech, Z. J., Havas, G., and Majewski, B. S. 1997. Perfect hashing. *Theor. Comput. Sci. 182,* 1-2, 1–143.

Dijkstra, E. W. 1960. Recursive programming. *Numer. Math. 2*, 312–318.

Dixon, R., McKee, T., Schweitzer, P., and Vaughan, M. 1989. A fast method dispatcher for compiled languages with multiple inheritance. In *Proc. OOPSLA'89*. ACM Press, 211–214.

Driesen, K. 1993a. Method lookup strategies in dynamically typed object-oriented programming languages. M.S. thesis, Vrije Universiteit Brussel.

Driesen, K. 1993b. Selector table indexing and sparse arrays. See OOPSLA [1993], 259–270.

Driesen, K. 2001. *Efficient Polymorphic Calls*. Kluwer Academic Publisher.

Driesen, K. and Hölzle, U. 1995. Minimizing row displacement dispatch tables. See OOPSLA [1995], 141–155.

Driesen, K., Hölzle, U., and Vitek, J. 1995. Message dispatch on pipelined processors. In *Proc. ECOOP'95*, W. Olthoff, Ed. LNCS 952. Springer, 253–282.

Ducournau, R. 1991. *Yet Another Frame-based Object-Oriented Language: YAFOOL Reference Manual*. Sema Group, Montrouge, France.

Ducournau, R. 1997. La compilation de l'envoi de message dans les langages dynamiques. *L'Objet 3,* 3, 241–276.

Ducournau, R. 2002a. Implementing statically typed object-oriented programming languages. Rapport de Recherche 02-174, LIRMM, Université Montpellier 2.

Ducournau, R. 2002b. La coloration pour l'implémentation des langages à objets à typage statique. In *Actes LMO'2002* in *L'Objet vol. 8*, M. Dao and M. Huchard, Eds. Lavoisier, 79–98.

Ducournau, R. 2002c. "Real World" as an argument for covariant specialization in programming and modeling. In *Advances in Object-Oriented Information Systems, OOIS'02 Workshops Proc.*, J.-M. Bruel and Z. Bellahsène, Eds. LNCS 2426. Springer, 3–12.

Ducournau, R. 2006. Coloring, a versatile technique for implementing object-oriented languages. Rapport de Recherche 06-001, LIRMM, Université Montpellier 2.

Ellis, G., Levinson, R. A., Fall, A., and Dalh, V., Eds. 1995. *Proc. of Int. Conf. on Knowledge Retrieval, Use, and Storage for Efficiency (KRUSE'95)*.

Ellis, M. and Stroustrup, B. 1990. *The annotated C++ reference manual.* Addison-Wesley, Reading, MA, US.

Fall, A. 1995. Heterogeneous encoding. See Ellis et al. [1995], 162–167.

Fall, A. 1998. The foundations of taxonomic encoding. *Computational Intelligence 14*, 598–642.

Gagnon, E. M. and Hendren, L. 2001. SableVM: A research framework for the efficient execution of Java bytecode. In *Proc. USENIX JVM'01*. 27–40.

Garey, M. and Johnson, D. 1979. *Computers and Intractability. A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, San Francisco (CA), USA.

Gil, J. and Zibin, Y. 2005. Efficient subtyping tests with PQ-encoding. *ACM Trans. Program. Lang. Syst. 27,* 5, 819–856.

Goldberg, A. and Robson, D. 1983. *Smalltalk-80, the Language and its Implementation.* Addison-Wesley, Reading (MA), USA.

Grove, D. and Chambers, C. 2001. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst. 23,* 6, 685–746.

Habib, M., Huchard, M., and Nourine, L. 1995. Embedding partially ordered sets into chain-products. See Ellis et al. [1995], 147–161.

Habib, M. and Nourine, L. 1994. Bit-vector encoding for partially ordered sets. In *ORDAL*, V. Bouchitté and M. Morvan, Eds. LNCS 831. Springer, 1–12.

Habib, M., Nourine, L., and Raynaud, O. 1997. A new lattice-based heuristic for taxonomy encoding. In *Proc. KRUSE'97*. 60–71.

Harbinson, S. P. 1992. *Modula-3.* Prentice Hall.

Huchard, M. and Leblanc, H. 2000. Computing interfaces in Java. In *Proc. of IEEE Int. Conf. on Automated Software Engineering (ASE'2000)*. 317–320.

Jensen, T. R. and Toft, B. 1995. *Graph Coloring Problems.* John Wiley.

Kiczales, G., des Rivières, J., and Bobrow, D. 1991. *The Art of the Meta-Object Protocol.* MIT Press.

Klefstad, R., Krishna, A., and Schmidt, D. 2002. Design and performance of a modular portable object adapter for distributed, real-time, and embedded CORBA applications. In *Proc. 4th Int. Symp. on Distributed Objects and Applications*. OMG.

Knuth, D. E. 1973. *The art of computer programming, Sorting and Searching.* Vol. 3. Addison-Wesley.

Krall, A. and Grafl, R. 1997. CACAO - a 64 bits JavaVM just-in-time compiler. *Concurrency: Practice ans Experience 9,* 11, 1017–1030.

Krall, A., Vitek, J., and Horspool, R. 1997. Near optimal hierarchical encoding of types. In *Proc. ECOOP'97*, M. Aksit and S. Matsuoka, Eds. LNCS 1241. Springer, 128–145.

Lippman, S. 1996. *Inside the C++ Object Model.* Addison-Wesley, New-York.

Liskov, B., Curtis, D., Day, M., Ghemawat, S., Gruber, R., Johnson, P., and Myers, A. C. 1995. THETA reference manual. Technical report, MIT.

Mehlhorn, K. and Tsakalidis, A. 1990. Data structures. See Van Leeuwen [1990], Chapter 6, 301–341.

Meyer, B. 1992. *Eiffel: The Language.* Prentice-Hall.

Meyer, B. 1997. *Object-Oriented Software Construction*, second ed. Prentice-Hall.

Meyer, J. and Downing, T. 1997. *JAVA Virtual Machine.* O'Reilly.

Microsoft. 2001. C# Language specifications, v0.28. Technical report, Microsoft Corporation.

Morris, R. 1968. Scatter storage techniques. *Commun. ACM 11,* 1, 38–44.

Mössenböck, H. 1993. *Object-Oriented Programming in Oberon-2.* Springer.

Muthukrishnan, S. and Muller, M. 1996. Time and space efficient method lookup for object-oriented languages. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*. ACM/SIAM, 42–51.

Myers, A. 1995. Bidirectional object layout for separate compilation. See OOPSLA [1995], 124–139.

Odersky, M., Spoon, L., and Venners, B. 2008. *Programming in Scala, A comprehensive step-by-step guide*. Artima.

OOPSLA 1993. *Proceedings of the 8th ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'93*. SIGPLAN Notices, 28(10). ACM Press.

OOPSLA 1995. *Proceedings of the 10th ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'95*. SIGPLAN Notices, 30(10). ACM Press.

OOPSLA 1997. *Proceedings of the 12th ACM Conference on Object-Oriented Programming, Languages and Applications, OOPSLA'97*. SIGPLAN Notices, 32(10). ACM Press.

Palacz, K. and Vitek, J. 2003. Java subtype tests in real-time. See Cardelli [2003], 378–404.

Pfister, B. H. C. and Templ, J. 1991. Oberon technical notes. Tech. Rep. 156, Eidgenossische Techniscle Hochschule Zurich–Departement Informatik.

Privat, J. 2006. PRM, the language. version 0.2. Rapport de Recherche 06-029, LIRMM, Université Montpellier 2.

Privat, J. and Ducournau, R. 2005. Link-time static analysis for efficient separate compilation of object-oriented languages. In *ACM Workshop on Prog. Anal. Soft. Tools Engin. (PASTE'05)*. 20–27.

Pugh, W. and Weddell, G. 1990. Two-directional record layout for multiple inheritance. In *Proc. PLDI'90*. ACM SIGPLAN Notices, 25(6). 85–91.

Pugh, W. and Weddell, G. 1993. On object layout for multiple inheritance. Tech. Rep. CS-93-22, University of Waterloo.

Queinnec, C. 1998. Fast and compact dispatching for dynamic object-oriented languages. *Information Processing Letters 64,* 6, 315–321.

Raynaud, O. and Thierry, E. 2001. A quasi optimal bit-vector encoding of tree hierarchies. application to efficient type inclusion tests. In *Proc. ECOOP'2001*, J. L. Knudsen, Ed. LNCS 2072. Springer, 165–180.

Schmidt, D. C. 1990. GPERF: A perfect hash function generator. In *USENIX C++ Conference*. 87–102.

Schubert, L., Papalaskaris, M., and Taugher, J. 1983. Determining type, part, color and time relationship. *Computer 16*, 53–60.

Shalit, A. 1997. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley.

Sprugnoli, R. 1977. Perfect hashing functions: a single probe retrieving method for static sets. *Comm. ACM 20,* 11, 841–850.

Steele, G. 1990. *Common Lisp, the Language*, Second ed. Digital Press.

Stroustrup, B. 1998. *The C++ programming Language, 3^e ed.* Addison-Wesley.

Taft, S. T., Duff, R. A., Brukardt, R. L., Ploedereder, E., and Leroy, P., Eds. 2006. *Ada 2005 Reference Manual: Language and Standard Libraries*. LNCS 4348. Springer.

Takhedmit, P. 2003. Coloration de classes et de propriétés : étude algorithmique et heuristique. M.S. thesis, Université Montpellier 2.

Tarjan, R. E. and Yao, A. C. C. 1979. Storing a sparse table. *Comm. ACM 22,* 11, 606–611.

Van Leeuwen, J., Ed. 1990. *Algorithms and Complexity*. Handbook of Theoretical Computer Science, vol. 1. Elsevier, Amsterdam.

Vitek, J., Horspool, R., and Krall, A. 1997. Efficient type inclusion tests. See OOPSLA [1997], 142–157.

Vitter, J. S. and Flajolet, P. 1990. Average-case analysis of algorithms and data structures. See Van Leeuwen [1990], Chapter 9, 431–524.

Warren, H. S. 2003. *Hacker's Delight*. Addison-Wesley.

Wirth, N. 1988. The programming language Oberon. *Software Practice & Experience 18,* 7, 671–690.

Zendra, O., Colnet, D., and Collin, S. 1997. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. See OOPSLA [1997], 125–141.

Zibin, Y. and Gil, J. 2003a. Incremental algorithms for dispatching in dynamically typed languages. In *Proc. POPL'03*. ACM, 126–138.

Zibin, Y. and Gil, J. 2003b. Two-dimensional bi-directional object layout. See Cardelli [2003], 329–350.

Received December 2005; revised December 2006, August 2007; accepted January 2008

A. ALGORITHMS FOR PERFECT HASHING

The computation of perfect hashing parameters is presented in a simple Common Lisp code [Steele 1990]. The entry is a list of all different integers and the output the H parameter. These integers are presumed to be the set of identifiers of C superclasses, but there is no other connection with object-oriented languages.

A.1 Modulus

The algorithm for computing H_C in the modulus case is quite straightforward and could likely be improved. Starting from n_C, the length of the list, each integer is tested to determine if it gives an injective hashing function h_C.

```
(defun perfect-hash-mod (ln)   ;; LN is a list of positive integers
  (if (null ln)                ;; with Java interfaces, LN may be NIL
      1
    (loop for n from (length ln) by 1
          until (ph-p ln n #'(lambda (x) (mod x n)))
          finally return n)))
```

Injectivity is checked by building a hashset—an array—and hashing all numbers until a collision occurs:

```
(defun ph-p (ln size fn)              ;; SIZE is the hashset size
  (let ((ht (make-array size)))       ;; FN is the hashing function
    (loop for i in ln
          do (let ((hv (funcall fn i)))
               (if (aref ht hv)       ;; test for a first collision
                   (return nil)
                 (setf (aref ht hv) i)))
          finally return t)))
```

The algorithms for quasi-perfect hashing are exactly the same as perfect hashing algorithms, apart from the function for testing injectivity which checks for a second collision instead of the first one:

```
(defun qph-p (ln size fn)              ;; SIZE is the hashset size
  (let ((ht (make-array size)))        ;; FN is the hashing function
    (loop for i in ln
          do (let ((hv (funcall fn i)))
               (if (cdr (aref ht hv))  ;; test for a second collision
                   (return nil)
                 (push i (aref ht hv))))
          finally return t)))
```

A.2 Bit-wise and

The same scheme would also apply to bit-wise `and`, by only changing `(mod x n)` into `(logand x (1- n))`. However, a slightly more sophisticated algorithm is far more efficient. One first computes all discriminant bits, i.e. bits which are not the same in all numbers. The resulting integer gives a perfect hashing function since all integer pairs differ by at least one bit. Then one checks each 1-value bit, by decreasing weight, and one switches the bit when it is not required for injectivity.

```
(defun perfect-hash-and (ln)     ;; LN is a list of positive integers
  (if (null (cdr ln))
      1
    (let ((mask (logxor (apply #'logior ln) (apply #'logand ln))))
                          ;; MASK contains only discriminant 1-bits
      (loop for b from (1- (integer-length mask)) by 1 downto 0
            when (logbitp b mask) do                  ;; B-bit is 1
            (let ((new (logxor mask (ash 1 b))))
                                ;; NEW is MASK with switched B-bit
              (when (ph-p ln (1+ new) #'(lambda (x) (logand x new)))
                 (setf mask new)))
            finally return (1+ mask)))))
```

In the previous code, `logxor`, `logior` and `logand` are COMMON LISP integer functions for bit-wise operations: exclusive and inclusive `or`, and `and`. `(Integer-length n)` gives the position of the leftmost 1-bit of a positive integer `n`. `(Logbitp n b)` tests if the `b`-th bit of `n` is 1. `(Ash n b)` shifts `n` left by `b` positions (when `b` is positive).

A.3 Complexity

Let H_C be the perfect hashing function result, n_C be the length of `ln`, and id_C the maximum value in `ln`. Assuming that all operations are time-constant, the complexity of `perfect-hash-mod` is roughly $H_C(H_C - n_C)$. The H_C factor is due to the array initializations and the actual computation only requires $n_C(H_C - n_C)$ operations. As H_C is bounded by $id_C + 1$, the function can be modified in order to allocate a single array of size $id_C + 1$ and reinitialize only n_C entries, instead of H_C, at each try. So, the worst-case complexity is $\mathcal{O}(n_C\, id_C)$.

The complexity of `perfect-hash-and` is roughly $id_C \log id_C$, since `mask` is bounded by $2id_C$. Again, the id_C factor is due to the array initialization and the actual computation only requires $n_C \log id_C$ operations. So, for the same reasons as for `perfect-hash-mod`, the worst-case complexity is $\mathcal{O}(id_C + n_C \log id_C)$.

However, on average, allocating an array of maximal size, i.e. $id_C + 1$ or `mask`, at each class loading appears to be less efficient than allocating a smaller array at each try. So, a better solution might be to allocate a single static array for all class loadings, and to enlarge it each time an overflow occurs. Overall, the complexity of PH-`and` seems to be better than that of PH-`mod`. This is true for worst-case complexities. However, on average, H_C is close to $2n_C$ for PH-`mod` and the PH-`mod` complexity is likely closer to n_C^2. In practice, without memory allocation, both functions take roughly 1 second for hashing the whole set of benchmarks presented in Table III, i.e. 37726 classes, hence about 26μs per class.

B. NOTE ON BENCHMARKS

Many people have contributed to the benchmarks used for our tests, including Karel Driesen and Jan Vitek. At the beginning of this work, a repository was Yoav Zibin's web site, `http://www.cs.technion.ac.il/~zyoav/`.

Each benchmark consists of a list of class descriptions. Each class description is itself composed of the class name, a list of superclass names—which does not distinguish the JAVA `implements` and `extends` relationships—a list of attribute names (unused here), and a last one for method names. Method names are *mangled* in order to deal with JAVA, C++ or C# static overloading.

Besides this relative simplicity, these benchmarks are not homogeneous—they are extracted from class libraries written in different languages and they must have dropped many non-orthodox features of these languages—e.g. `virtual` or non-`virtual` inheritance in C++, multiple inheritance of the same class in EIFFEL, JAVA interfaces, etc. So, they must not be confused with the original programs.

Moreover, their origin may no longer be known and they may have been extracted from actual programs with another goal and other non-syntactic conventions. So some data may be suspected or even absent. For instance, many benchmarks have been established for subtyping tests—so, they do not include any information about attribute and method numbers. In contrast, attribute and method numbers might include `static` properties, though they are absolutely not concerned by object-oriented implementation. In some benchmarks, the method numbers are biased by the fact that only polymorphic methods, i.e. methods with at least two definitions, are considered. For instance, this bias makes our estimation overly pessimistic and this probably explains the relatively bad ratios for JAVA benchmarks in full multiple inheritance (Table IV). Anyway, with full multiple inheritance, the bias is limited and it did not require any particular action.

Benchmarks for JAVA. In contrast, testing JAVA techniques is more complicated. Apart from variants of the Java1.6 benchmarks, there is no distinction between classes and interfaces, i.e. all supertypes in Java benchmarks, interfaces and classes alike, are counted as superclasses in full multiple inheritance. As this distinction had been dropped, we computed interfaces according to a heuristic which gives a good approximation. Roughly speaking, this heuristic amounts to examining all diamond situations alike Figure 2 and considering that: (1) A is an interface, unless it is `java.lang.object`, (2) B or C must be an interface and (3) the other one, together with D, can be a class, unless otherwise constrained. More technically, this means that the interfaces must form an *edge-dominating set* of the *conflict graph* considered in coloring (see Note 7, page 7). This heuristic tends to minimize the actual number of interfaces to those which are necessary to make a valid JAVA hierarchy—hence, the original JAVA hierarchy may have many more interfaces.

Another point comes from *anonymous inner classes*—a JAVA idiom allows the programmer to implement an interface while instantiating the implementing class. It follows that, firstly, these anonymous classes do not appear in most benchmarks because this would require parsing the complete method code, instead of staying at a higher level, and secondly, some interfaces seem to be unimplemented because they are actually only implemented by anonymous classes. A related issue concerns *abstract classes*—i.e. classes without proper instances. These classes do not need

any physical implementation, but all benchmarks have dropped the information.

The last issue concerns transitivity edges—they all have been removed from the original benchmarks. However, it would be meaningful for the aforementioned heuristic to keep the often implied relation between a class and `java.lang.object`, when the considered class does not explicitly `extend` any particular class, but `implements` an interface, which is itself a subtype of `java.lang.object`. It follows that, in the type hierarchy, some types can be considered as a direct subclass of `java.lang.object` or as a non-implemented interface. Our heuristic chooses the former situation.

Altogether, we attempted to check that our heuristics did not introduce too much bias—actually, we consider ratios and the precise interface identification changes both numerators and denominators, in a roughly proportional way. In order to check it, we created a new benchmark[21], Java1.6, extracted from JDK 1.6, which includes both information on interfaces and abstract classes. For assessing the bias of different approaches, we consider the following variants: (1) the original class/interface labelling (about 1000 interfaces), (2) our heuristic which identifies about 250 interfaces, (3) the original labelling but in considering that non-implemented interfaces are classes (about 500 interfaces), (4) the same but only non-implemented leaves are considered as classes and, finally, (5) an extra class is added for implementing each leaf interface. Despite the high variation, within a 4 ratio, in interface number, the tests gave similar results on all parameters—e.g. size of `implements` relation, PH parameters—apart from DA which clearly depends on the interface number. In Section 5, xJava1.6 is variant (2), Java1.6 is variant (5) without abstract classes, and yJava1.6 is variant (5) with abstract classes. The variant (5) has been chosen because it is the least favourable for PH. When extrapolating these experiments to the other benchmarks, one may expect that the perfect hashing results are meaningful. In contrast, as the number of interfaces is highly underestimated, our benchmarks likely underestimate DA parameters—this confirms that DA must be ruled out even in its shared variant.

All Java benchmarks behave in a closely similar way, except IBM-SF. Instead of explaining this original behaviour by a possible original programming style, one might also consider an hypothetical artifact, in the benchmark itself or in the heuristic for computing interfaces. This would be a nice conclusion since IBM-SF is the only reason for not considering that perfect hashing is actually perfect for Java.

[21] Thanks to Floréal Morandat at LIRMM.